Search-Based Generation of Human Readable Test Data and Its Impact on Human Oracle Costs

by

Sheeva Afshan

Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy at
The University of Sheffield
Department of Computer Science
March 2013

Abstract

The frequent non-availability of an automated oracle makes software testing a tedious manual task which involves the expensive performance of a human oracle. Despite this, the literature concerning the automated test data generation has mainly focused on the achievement of structural code coverage, without simultaneously considering the reduction of human oracle cost.

One source of human oracle cost is the unreadability of machine-generated test inputs, which can result in test scenarios that are hard to comprehend and time-consuming to verify. This is particularly apparent for string inputs consisting of arbitrary sequences of characters that are dissimilar to values a human tester would normally generate. The key objectives of this research is to investigate the impact of a seeded search-based test data generation approach on test data oracle costs, and to propose a novel technique that can generate human readable test inputs for string data types.

The first contribution of this thesis is the result of an empirical study in which human subjects are invited to manually evaluate test inputs generated using the seeded and unseeded search-based approaches for 14 open source case studies. For 9 of the case studies, the human manual evaluation was significantly less time-consuming for inputs produced using the seeded approach, while the accuracy of test input evaluation was also significantly improved in 2 cases.

The second contribution is the introduction of a novel technique in which a natural language model is incorporated into the search-based process with the aim of improving the human readability of generated strings. A human study is performed in which test inputs generated using the technique for 17 open source case studies are evaluated manually by human subjects. For 10 of the case studies, the human manual evaluation was significantly less time consuming for inputs produced using the language model. In addition, the results revealed that accuracy of test input evaluation was also significantly enhanced for 3 of the case studies.

Acknowledgement

First and foremost, I would like to acknowledge the enthusiastic and supportive guidance of my supervisor, Dr Phil McMinn. His helpful discussions and extensive knowledge of this subject has provided the means to develop new approaches to solve problems. I am indebted to Dr Mark Stevenson for providing the source code of the language model and for his technical assistance. Also, I would like to thank Dr Neil Walkinshaw for his useful comments on the thesis.

I would like to say a special thank you to all the people in my research group (Verification and Testing) who have provided help during my studies, and for completing my questionnaires and their beneficial feedback.

I am also thankful to the project sponsor. This work would not have been possible without the financial support, EPSRC, for which I am eternally grateful.

Finally, I would like to thank my beloved parents, my dear auntie and my fiancé for their understanding, endless patience, and constant encouragement.

Publications

- S.Afshan, P. McMinn and M. Stevenson. Searching for Readable String Test Inputs using a Natural Language Model to Reduce Human Oracle Cost. In Proceedings of the International Conference on Software Testing, Verification and Validation (ICST 2013). IEEE Computer Society.
- S.Afshan and P. McMinn. An Investigation into Qualitative Human Oracle Costs. Proceedings of the Psychology of Programming Interest Group Annual Workshop (PPIG 2011).

Contents

1 Introduction **1**

1.1 Overview 1
1.2 The Topic Explored in This Thesis 2
1.3 Overall Research Aims and Objectives 3
1.4 Research Hypotheses 4
1.5 Contributions of this Thesis 6
1.6 Overview of The Structure of The Thesis 7

2 Literature Review **9**

2.1 Introduction 9
2.2 Structural (White-Box) Testing 9
2.2.1 Basic Concepts 11
2.2.2 Random Testing 13
2.2.3 Symbolic Execution 13
2.3 Meta-heuristic Search Techniques 14
2.3.1 Hill Climbing 16
2.3.2 Simulated Annealing 16
2.3.3 Evolutionary Algorithms 18
2.4 Search-Based Test Data Generation 23
2.4.1 Fitness Function for Branch Coverage 24
2.4.2 IGUANA 25
2.4.3 Applying Alternating Variable Method 27
2.4.4 Applying (1+1) Evolutionary Algorithm 28
2.4.5 Search-Based Test Data Reduction Techniques 30
2.4.6 Seeded Search-Based Techniques 31

2.5 Test Data Evaluation 35
2.5.1 Pseudo Oracle 36
2.5.2 Specification-Based Oracle 38
2.5.3 Invariant-Based Oracle 39
2.5.4 Metamorphic Based Oracle 39
2.5.5 Consistency Oracle 41
2.5.6 Heuristic Oracle 41
2.5.7 Human Oracle 42
2.6 Mutation Analysis 43
2.7 Software Engineering Empirical Studies 46
2.7.1 Human Empirical Studies 47
2.7.2 Crowd-Sourcing in Empirical Studies 49
2.8 Conclusions 52

3 An Investigation into a Seeded Search-Based Approach For Branch Coverage 53
3.1 Introduction 53
3.2 The Search-Based Technique 54
3.3 Experimental Study Methodology 55
3.3.1 Case Studies 56
3.3.2 Human Study Protocol 58
3.3.3 Participant Selection 59
3.3.4 Generating Test Inputs 63
3.3.5 Basic Definitions 64
3.3.6 Research Questions 65
3.4 Experimental Results 66
3.5 Threats to Validity 82
3.6 Conclusions 85

4 An Investigation into a Seeded Search-based Approach For Oracle Cost 86
4.1 Introduction 86
4.2 Experimental Study Methodology 87
4.2.1 Test Input Selection 88

4.2.2 Human Study Protocol 88
4.2.3 Participant Selection 90
4.2.4 Usable Judgements 91
4.2.5 Basic Definitions 91
4.2.6 Research Questions 92
4.3 Experimental Results 92
4.4 Threats to Validity 99
4.5 Conclusions 100

5 Test Data Generation Using A Language Model 102
5.1 Introduction 102
5.2 Language Models 104
5.3 Incorporating a Language Model Into Search-Based Test Input Generation 107
5.4 Experimental Study Methodology 109
5.4.1 Case Studies 110
5.4.2 Generating String Test Inputs 113
5.4.3 Test Input Selection 114
5.4.4 Human Study Protocol 114
5.4.5 Participant Selection 115
5.4.6 Usable Judgements 116
5.4.7 Research Questions 116
5.5 Experimental Results 117
5.6 Threats to Validity 127
5.7 Conclusions 128

6 Conclusions and Future Work 130
6.1 Summary of Achievements 130
6.1.1 Hypotheses 130
6.1.2 Contributions of this Thesis 132
6.2 Summary of Future Work 133
6.2.1 Investigating Various Seeding Schemes 133
6.2.2 Managing Fault-Finding Capability 133
6.2.3 Improving Readability 134

6.2.4 Test Input Generation for Various Data Types 135

Bibliography **137**

Chapter 1

Introduction

1.1 Overview

An important development in Software Engineering Research was the introduction of what became known as "Search-Based Software Testing" in 1976 by Miller and Spooner [88]. The approach was aimed to apply search-based optimisation techniques to automate or partially automate software testing. However, despite several improvements to the technique over the last few decades, the uptake in industry has been low and software testing is still a laborious and expensive process.

Test data evaluation is a critical component of the software testing process that monopolise a large proportion of software testing budgets. Without this phase, software testing fails to achieve its fundamental objective of revealing the system's failures or ensuring it operates as expected. This process is however subject to the existence of an automated mechanism, commonly referred to as an *oracle*, that can determine the expected behaviour of the program under test (Figure 1.1) [87]. As a results, test data evaluation is performed manually by human testers for many real applications. This problem is commonly referred to as the *oracle problem*, and has been a challenging issue for several decades.

In addition, test data generation using the conventional search-based approaches often leads to extensive sets of arbitrarily looking values that are difficult to comprehend from a human perspective. Manual evaluation

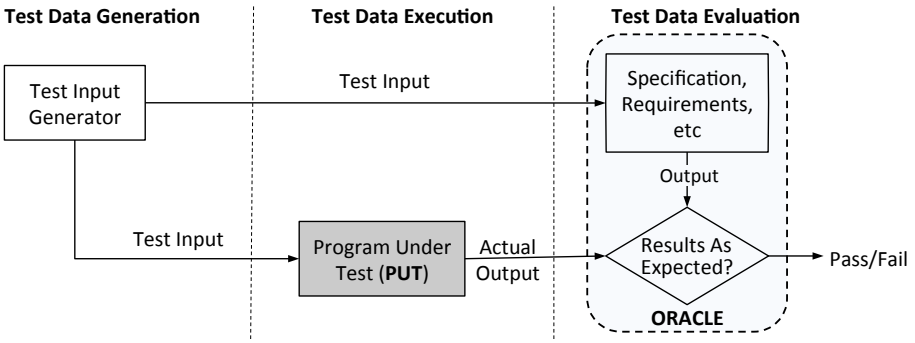


Figure 1.1: The essential phases of software testing: test data generation, test data execution and test data evalution

of these test inputs can be a tedious, error-prone and time consuming task. The main objective of this thesis is to propose and develop techniques that can reduce the human oracle effort associated with automatically generated test inputs.

1.2 The Topic Explored in This Thesis

Conventional approaches to automatic test data generation [45, 54, 79] tend to produce large volumes of arbitrarily looking and difficult-to-read inputs as long as branch coverage (a common objective for structural testing) is obtained. However, the output of a program, when executed with a generated set of test inputs, must still be evaluated for its correctness. A human oracle is often required to comprehend the test scenarios encoded by such inputs in order to decide wether the program's output is as expected. This forms a significant cost frequently referred to as *human oracle cost* [51, 82].

One source of human oracle cost originates from the quantity of the automatically generated test cases, and consequently the corresponding amounts of oracle data required for comparison. Automatically generated test suites often contain a large number of test cases that satisfy the same testing objectives. Test suite reduction techniques [126] can be applied to condense the number of test cases in a test set, and also reduce the relevant evaluation costs. A complete description of these techniques is presented in Section 2.4.5.

Another major source of oracle cost is the difficulty of reading these test cases. In particular, string values generated by automatic test input generators often appear as arbitrary sequences of characters such as "*f#p%F@}UM%5.*6ZY*" for an email address, rather than natural, instantly-readable strings such as "*James@gmail.com*". This results in test scenarios that are difficult to interpret and test cases that are time-consuming to manually evaluate. This effort adds a cost to the testing process, referred to as the *qualitative human oracle cost* [82], and is the main topic of this thesis.

Seeded search-based test data generation approaches can produce readable test cases by incorporating additional knowledge into the search mechanism [42, 82, 125]. This knowledge is often in the form of readable test cases that can be used as seeds to commence the search mechanism. These values can be collated from various resources including program's source code, specifications, code comments [82], or the programmer themself. A recent approach sources readable strings from human-created web pages using automated web queries [84, 105]. This approach, however, requires the program to have useful identifiers that can be reformulated into web search queries, otherwise pages containing suitable strings may not be found.

The application of the seeded search-based approaches depend upon a ubiquitous resource that can supply sample of test inputs as the starting seeds. Due to unavailability of such a resource, the ultimate objective of this research is to propose and develop a novel technique that can automatically assess and improve the readability of the potential values for string test inputs. Prior to this, this research inspects the effects of a seeded search-based test data generation approach on human oracle costs. More details about this is presented as follows.

1.3 Overall Research Aims and Objectives

The key objective of this thesis is to review the qualitative aspects of human oracle costs, investigate and establish methods that can effectively reduce these costs. Firstly the effectiveness and efficiency of a seeded search-based test data generation approach in producing branch-covering, fault-revealing and readable test inputs is inspected. Due to the limitations of the seeded

search-based approach, a new technique is then introduced for generating branch-covering readable values for string inputs. This approach incorporates a statistical language model into the search-based test data generation mechanism that can assess and improve the readability of the potential test inputs. The common aims and objectives of this research can therefore be summarised as follows:

1. To assess the efficiency and effectiveness of a seeded search-based approach as a technique for producing branch-covering fault-revealing test inputs.

2. To assess the effectiveness of the seeded search-based test data generation approach on test data readability, and its impact on human oracle costs.

3. To develop and evaluate a novel approach in which test data generation is integrated with a statistical language model to generate readable branch-covering values for string inputs, and to assess its impact on human oracle costs.

1.4 Research Hypotheses

This section sets the overall objectives of this thesis into different research hypotheses and describes each in detail. Each hypothesis is then be treated individually, in separate chapters, with a summary at the end.

Hypothesis 1 *Seeding the search-based test data generation process with human-supplied test inputs can produce test data with higher branch coverage, and without any detrimental effects on fault-finding effectiveness.*

To investigate this hypothesis, a search-based test data generation approach is seeded with samples of test inputs collated from human subjects for a number of Java programs. The test data generated using this approach is then compare with those generated using the standard (unseeded) approach in terms of branch coverage and fault-finding capabilities. The efficiency and effectiveness of each approach is also assessed based on the number of

fitness evaluations performed and their success rate in covering individual branches.

A similar approach was conducted by Fraser et al [42] for object oriented programs using the test inputs originated from programmers. Alshahwan et al [15] also proposed a seeded search-based strategy for testing web applications using the values collected dynamically from the web pages. In this thesis, the sample test inputs are collated from human subjects via a crowd-sourcing platform.

The key purpose behind using seeding in this thesis is to reduce human oracle costs. The Alternating Variable Method (AVM) [71] was chosen as a local search method to ensure the final test data retains readable characteristics of the seeded inputs.

Hypothesis 2 *Seeding the search-based test data generation process with human-supplied test inputs can produce readable test data that are less time-consuming and less error-prone for manual evalution.*

To investigate this hypothesis, the oracle costs for test data generated using both the seeded and unseeded search-based approaches are estimated using a human empirical study. Human subjects are recruited to manually evaluate test cases generated using each approach, while being timed during the process. The time and accuracy of subjects in evaluating test cases of each approach is used as a measure for test data readability and the corresponding oracle costs.

Hypothesis 3 *Incorporating the search-based test data generation process with a statistical language model can produce more readable test data for string variables, which are less time-consuming and less error-prone for manual evalution.*

To investigate this hypothesis, the search-based test data generation process is incorporated with a language model that can estimate the probability of a string occurring in a natural language. This probability score can be viewed as a measure for "likeness" or similarity of a string to naturally occurring words and thus can be used to guide the search towards more natural

and inherently readable string inputs.

The effectiveness of this approach in generating readable branch-covering string inputs and its real impact on human oracle costs are assessed using an human empirical study. Human subjects are recruited to manually evaluate test inputs generated using both the language model and the conventional search-based approaches, while being timed. The time and accuracy of subjects in evaluating test cases of each approach is assessed and compared. The effects of this approach on test data fault-finding effectiveness is also evaluated.

1.5 Contributions of this Thesis

The contributions of this thesis are as follows:

1. The results of an empirical study in which a seeded search-based approach is implemented and compared against a conventional unseeded search-based approach for generating branch-covering and fault-detecting test inputs. The analysis reveals cases in which the seeded approach outperforms the unseeded approach in terms of branch coverage, efficiency, and fault- finding effectiveness.

2. The results of a human empirical study in which test data generated using the seeded and unseeded search-based approaches are evaluated by human subjects. The analysis reveals cases in which test data generated using the the seeded approach is both less time consuming and less error-prone to manually evaluate by human subjects.

3. The introduction of a technique that incorporates the search-based mechanism with a statistical language model to automatically generate readable branch-covering values for string inputs.

4. The results of a human study in which the language model technique is compared with the conventional search-based approach. The analysis reveals cases in which test inputs generated using the language model

approach are both less time consuming and error-prone to manually evaluate by human subjects.

1.6 Overview of The Structure of The Thesis

This thesis is organised as follows:

Chapter 2 – Literature Review explores the literature in the field of search-based structural testing. The chapter begins by describing a number of search-based techniques employed in automated test data generation, including Hill Climbing, Simulated Annealing and Evolutionary Algorithms. It then discusses some of the major issues associated with search-based test data generation such as the size (quantity) and readability (quality) of generated test suites and how these aspects can affect the overall testing costs. The chapter then proceeds to describe the oracle problem and the various types of oracles, discussing how different aspects of automatically generated test data can particularly impact the human oracle costs. This is then followed by a discussion on various empirical studies in software engineering and an investigation into mutation testing and mutation analysis as two major evaluation methods employed in this thesis.

Chapter 3 – An Investigation into a Seeded Search-Based Approach For Branch Coverage and Fault Finding Capability presents and analyses the results of an empirical study in which samples of test cases are gathered from human subjects for a number of Java method. These test cases are then used as seeds to start the automatic test input generation process. The seeded search-based approach is then compared against the unseeded convectional approach with respect to the branch coverage and fault-finding effectiveness of the test data these generate.

Chapter 4 – An Investigation into a Seeded Search-based Approach For Oracle Cost presents another empirical study in which human subjects are recruited to manually evaluate test cases generated using both the seeded and unseeded search-based approaches by hand, while be-

ing timed during the process. Human subjects are expected to provide the correct outputs of a number of Java method for a presented set of test cases. The main objective of this chapter is to assess the time human subjects would require to manually evaluate test cases of each approach, and to investigate whether test data generated using the seeded search-based approach are less time consuming and less error-prone to evaluate.

Chapter 5 – Test Data Generation Using A Language Model introduces a new approach in which a language model is incorporated into the search-based test data generation process to encourage generation of readable values for string inputs. The language model assigns a probability score to a string reflecting its likelihood occurring in a natural language. This chapter describes how this score can be used to form an additional component of search-based data generation for producing branch-covering readable string inputs. The technique is then empirically assessed using a human study. Human subjects are recruited and requested to manually evaluate test data generated using both this approach and the conventional search-based approach. The main objective of this investigation is to evaluate wether the incorporation of a language model in search-based test data generation can significantly improve readability of the string test inputs.

Chapter 6 – Conclusions and Future Work concludes the main body of the thesis with final comments and avenues for future work.

Chapter 2

Literature Review

2.1 Introduction

This chapter reviews the literature in the field of search-based test data generation and discusses some of the common problems associated with test data evaluation.

The chapter firstly describes the basic concepts of structural testing. It then investigates various search-based approaches to structural test data generation focusing on two commonly used search-based algorithms; evolutionary algorithms and hill climbing. Particular attention is paid to a local search method called the "Alternating Variable Method", first introduced by Korel [71] and the (1 + 1) Evolutionary Algorithm [118], which are the two test input generation algorithms, specifically used in this thesis.

The chapter then describes mutation analysis and reviews various human empirical studies in Software Engineering as the two main evaluation schemes employed in this thesis. Table 2.1 details where each subsection of this chapter is of particular relevance to a subsequent chapter.

2.2 Structural (White-Box) Testing

Structural test data generation is the process of deriving test cases from the internal structure of the program under test (PUT). This section describes various approaches to structural testing, summarising some of the main achievements in automated structural test data generation.

Table 2.1: Shows where each subsection of the literature review is relevant to the remaining chapters

Chapter 2 (Literature Review)	**Chapter 3**	**Chapter 4**	**Chapter 5**
2.2 Structural (White Box) Testing	3.3 Experimental Study Methodology		5.4 Experimental Study Methodology
2.3 Meta heuristic Search Techniques	3.3 Experimental Study Methodology		5.4 Experimental Study Methodology
2.3.1 Hill Climbing	3.3.4 Generating Test Inputs		
2.3.3 Evolutionary Algorithms			5.4.2 Generating String Test Inputs
2.4 Search-Based Test Data Generation	3.3 Experimental Study Methodology		5.4 Experimental Study Methodology
2.4.3 Applying Alternating Variable Method	3.3.4 Generating Test Inputs		
2.4.1 Fitness Function	3.3.4 Generating Test Inputs		5.4.2 Generating String Test Inputs
2.4.2 IGUANA	3.3.4 Generating Test Inputs		5.4.2 Generating String Test Inputs
2.5 Test Data Evaluation		4.2 Experimental Study Methodology	5.4 Experimental Study Methodology
2.5.7 Human Oracle		4.2.2 Human Study Protocol	5.4.4 Human Study Protocol
2.6 Mutation Analysis	3.4 RQ4- Test Data Fault Finding Capability		5.5 RQ4- Test Data Fault Finding Capability
2.7 Software Engineering Empirical Studies	3.3 Experimental Study Methodology	4.2 Experimental Study Methodology	5.4 Experimental Study Methodology
2.7.1 Human Empirical Studies	3.3.2 Human Study Protocol	4.2.2 Human Study Protocol	5.4.4 Human Study Protocol
2.7.2 Use of Crowd sourcing in Empirical Studies	3.3.3 Participant Selection	4.2.3 Participant Selection	5.4.5 Participant Selection

```
1  boolean isValidProtocol(String protocol)
2  {
3    int len = protocol.length();
4    if (len < 1)
5       return false;
6    char c = protocol.charAt(0);
7    if (!Character.isLetter(c))
8       return false;
9    for (int i = 1; i < len; i++)
10   {
11    c = protocol.charAt(i);
12    if (!Character.isLetterOrDigit(c) &&
13      c != '.' && c != '+' && c != '-') {
14      return false;
15    }
16   }
17   return true;
18 }
```

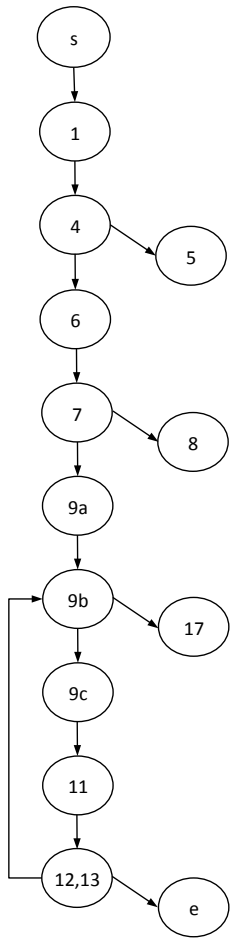


Figure 2.1: Code and control flow graph (CFG) of *isValidProtocol*

2.2.1 Basic Concepts

In structural testing, the internal structure of the PUT is identified using the program's *control flow graph (CFG)*. This is referred to the graphic representation of all paths that may be traversed through the program during its execution. A control flow graph for a program P is formally defined as $G = (N, E, V)$ where N is a set of nodes that represent the processing statements like definition, computation and predicates. E is a set of edges that represent the control flow between processing statements. V is a set of basic blocks including the start and end nodes. Each node $n \in N$ is a statement in the program, with each edge $e = (n_i, n_j) \in E$, representing a control transfer from node n_i to node n_j [39].

A sample CFG for the *isValidProtocol* program is represented in Figure 2.1. In this instance, N = {s, 1, 3, 4, 5, 6, 7, 9a, 9b, 9c, 17, 11, 12, 13, e} is a set of all the nodes, where nodes 9a, 9b, and 9c respectively represent the statements $i = 1$, $i < len$, and $i++$ in the for loop at line 9. Nodes 4, 7, 9b, and 12 are the branching nodes, and outgoing edges from these nodes are referred to as branches. V = {s, 5, 8, 17, e} is a set of basic blocks including the start node (s), and the exit node (e).

A program's control flow graph consists of the following structural elements:

1. A *statement*: refers to each declaration or assertion in the source code.
2. A *path*: refers to a path that is traversed from the start node to an end node through the program's execution.
3. A *branch*: refers to each boolean decision point in the code which leads to two structural elements: a true branch and a false branch.

Structural test data generation makes use of the information obtained from at least one of these structural elements. Structural testing for statement and path coverage requires generating test cases that cause the execution of all statements and all possible execution paths in the program during the course of testing.

Structural testing for branch coverage requires production of a sufficient number of test cases that invoke each entry point to the program or subroutine at least once, and can cover all the possible outcomes (true and false) of each branch at least once.

For instance, to achieve full branch coverage for the *isValidProtocol* program (displayed in Figure 2.1), it is necessary to develop a set of test cases that exercise all the branches in the code. As clear from the program's CFG, branches b_{4-5}, b_{7-8}, and b_{9b-17} are each covered when $protocol.length() < 1$, $!Character.isLetter(c)$, and $i > len$ respectively. Otherwise the final branch b_{13-14} is executed when all the remaining characters of the string variable *protocol*, starting from index 1 (i.e. the second character) are either alphanumeric or one of the characters {., +, -}. A set of appropriate test

cases for this example is therefore {"abcdtsg", "1bcdtsg"} which covers all the branches of this program.

2.2.2 Random Testing

Random testing is the process of generating test cases for a program at random. Targeting a predefined testing goal such as path or branch coverage, the approach attempts to iteratively generate test inputs at random until test inputs covering the specified path or branch are discovered. This approach often fails to generate suitable values for programs with complex branching structure due to the compound constraints of the desired path. In such scenarios, the randomisation scheme is unlikely to generate test inputs that cause the execution of a difficult-to-reach branch.

For example, the execution of the branch $if(a == b \;\&\&\; b == c)$ requires generating three equal input values for the three variables a, b and c. This branch is however very unlikely to be executed at random unless the size of the input domain is relatively small. As another example, consider the code fragment shown below:

```
1  if (args == ''LUCKY'') {
2   //Target
3  }
```

The probability that a randomly generated input for the variable $args$ will be equal to the string "$LUCKY''$ is very low. In such situations a more directed search technique that is capable of locating appropriate test data is required.

2.2.3 Symbolic Execution

Test data generation using symbolic execution requires computing values of the variables in the PUT as a set of functions that represents a sequence of operations. The sequence is accomplished as the execution is traced along a specific path through the program. Each function represents a series of constraints in the program describing the execution of a particular path. In this process, the program's inputs are represented as symbols and program outputs are expressed as mathematical expressions involving these symbols.

The state of a symbolically executed program includes the (symbolic) values of the program variables and a path condition (PC). The path condition is a Boolean formula over the symbolic inputs. This encodes all the constraints each input must satisfy in order to cause the execution of a particular path. The paths that are traversed during the symbolic execution of a program can be represented by a symbolic execution tree. For instance, symbolic execution of the *test_me* program of Figure 2.2, starts with these symbolic values: $x = X$, $y = Y$, where the initial value for the path condition is set to true.

As illustrated in the execution tree in Figure 2.2, at each branch point, the PC is updated with constraints on the inputs to select from the alternative paths. After executing line (1) in the code, both alternative paths of the if statement are achievable. If the path condition becomes false, the corresponding path is denoted as invalid, and the symbolic execution excludes that path. In this example, symbolic execution investigates three dissimilar valid paths and one invalid path (Path 3). For test case generation, the obtained path conditions are solved and the solutions are used as test inputs that are guaranteed to exercise all the paths through this code.

The development of programs for symbolic execution is very expensive due to the presence of loops and computed storage locations. This approach is therefore mainly used for testing numerical programs, where the cost/benefit relation is acceptable [23].

Dynamic symbolic execution was first introduced by Godefroid et al [46] to resolve some of the challenges faced by static symbolic execution. This approach, also known as concolic testing [104], aids symbolic execution by obtaining information through dynamic analysis of the program under test. In contrast to static testing, the principle of dynamic approach is to execute the program under test and to systematically explore all the feasible paths through the program in order to generate adequate test data.

2.3 Meta-heuristic Search Techniques

Meta-heuristic search techniques refer to methods that adopt heuristic mechanisms as the principal search strategies to solve computational problems

```
1 public void test_me()
2 {
3   int x, y;
4   if (x > y)
5   {
6     result = x - y;
7   }
8   else
9   {
10    result = y - x;
11  }
12 }
```

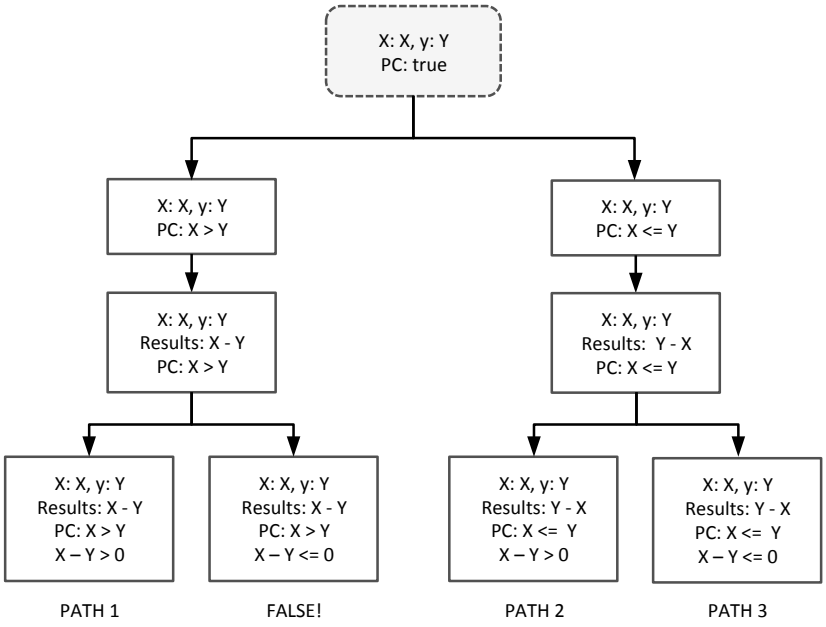


Figure 2.2: Symbolic execution tree for the *test_me* program . This program computes the difference between two integers x and y. Path Condition (PC) is the conjunction of all symbolic constraints along a path. After executing the first line, both alternative paths of the if statement are achievable. A false PC implies an invalid path (symbolic execution does not traverse that path). In this example, symbolic execution investigates three dissimilar valid paths and one invalid path (Path 3).

within a large search space. Meta-heuristic search techniques such as Hill Climbing [89], Simulated Annealing (SA) [114], and Evolutionary Algorithms (EA) [124] have been applied on a variety of testing problems including test data generation [38, 50, 54, 71, 78, 120]. This section firstly describes the operation of several search-based techniques and then investigates the application of these techniques on test data generation.

2.3.1 Hill Climbing

Hill Climbing is a well known local search algorithm that attempts to improve a single candidate solution, commencing from a random point in the search space. The neighbours of the current solution are investigated in the search space until a better solution is located or the resources are exhausted.

The improvement from one solution to another is completed using either *first ascent* or *random ascent* strategies. In the first ascent approach, all the neighbours of the current solution are investigated and the neighbour solution with the greatest improvement replaces the current solution. In the random ascent approach, the neighbours of the current solution are evaluated at random. The first improved value then replaces the current solution. Figure 2.3 presents a high level pseudo code for this algorithm [79].

This approach is called Hill Climbing for the reason that the underlying search space can be considered as a landscape with peaks representing points of higher fitness. The algorithm selects a hill near to the randomly chosen starting point and moves the current point to the top of this hill [79]. The hill located by the algorithm is however likely to be a local maxima as opposed to a global maxima. This is one of the disadvantages of this algorithm, referred to as the *local maxima problem*, and is demonstrated in Figure 2.4.

2.3.2 Simulated Annealing

Simulated Annealing (SA) [69, 114] is a global search algorithm that models the thermal process in which a heated metal freezes into a minimum energy crystalline structure. This process consists of the following steps:

```
Hill_Climbing()
s ← s ∈ S // choose an initial individual s uniformly
from the search space ;
repeat
    s′ ← s ∈ N(s) // select a new value from the
    neighbourhood of s;
    if (obj(s′) < obj(s)) then
        //replace the current value with the new one;
        s ← s′;
    end
until (termination codition);
return s;
```

Figure 2.3: Pseudo code illustrating the Hill Climbing algorithm, for a problem with solution s, search space S, neighbourhood structure N, and the objective function obj to be minimised.

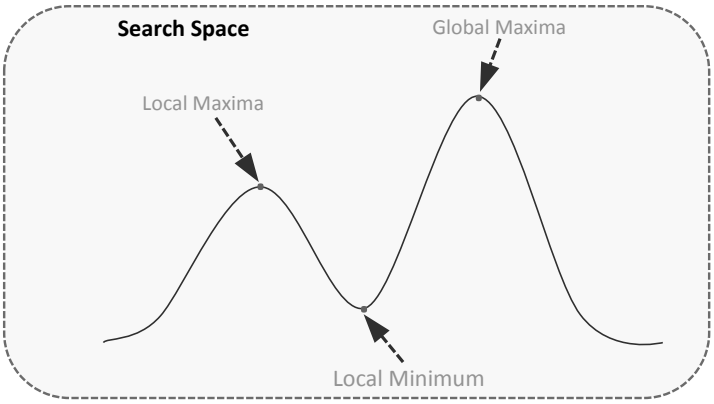


Figure 2.4: The local maxima problem in the Hill Climbing algorithm

1. Increasing the heat temperature to a maximum value – the metal melting point.

2. Slowly decreasing the temperature until the particles are arranged into a ground state of the metal (the annealing phase).

Simulated Annealing performs similarly to the Hill Climbing, with the difference that it allows movements to poorer solutions. This enables the search to have less restricted movements (downhill movements) within the search space. These movements are implemented during the search based on the following two parameters:

1. The objective value between two solutions.

2. The control parameter known as the temperature.

The temperature is initially high in order to allow free movements around the search space. As the search progresses, the temperature decreases according to a cooling schedule.

Figure 2.5 presents a high level pseudo code for this algorithm. The SA algorithm starts with a random solution s, and creates a new solution s' by adding small perturbation to the s. If the new solution is better ($obj(s') < obj(s)$), it replaces the current solution. Otherwise, SA applies a stochastic acceptance criterion based on a certain probability, which is controlled by the temperature parameter T and is reduced over time.

Simulated Annealing can overcome the local maxima problem when the local maximum is near the global maximum. In this case, one of the search movements can be diverted from the local maximum and reach the ascending slope of the global maximum. For a complete discussion the reader is referred to reference [79].

2.3.3 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are one of the most popular meta-heuristic search algorithms that are inspired by biological evolution: reproduction, mutation, recombination, and selection. Evolutionary Algorithms operate

```
Simulated_Annealing()
s ∈ S // s is a starting solution selected from the search space S;
i ← 0 ;
while (obj(s_i) > 0 && i < max_evaluations) do
    t ← CoolingSchedule(i) ;
    i ← i + 1 ;
    while (t > eps) do
        s' ← random_neighbor(s);
        ΔE = obj(s') − obj(s);
        if (ΔE < 0) then
            s ← s';
        else
            r ← random_number_between(0, 1);
            if (r < e^{−ΔE/t}) then
                s ← s';
            end
        end
    end
end
```

Figure 2.5: Pseudo code illustrating the algorithm for Simulated Annealing. s represents the initial solution which is iteratively altered while the stopping condition is unsatisfied. s' is a new solution generated at random in the neighbourhood of s. t shows temperature level for each solution, *eps* represents the lowest temprature. *CoolingSchedule* is a decrement function for lowering the temperature (t), and *obj* is the objective function to be iteratively minimised within the specified range of fitness evalutions (*max_evaluations*).

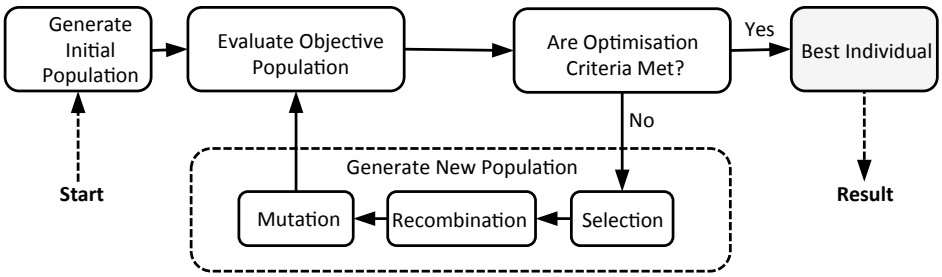


Figure 2.6: Different phases of a GA

based on iterative searching processes resulting in a 'population' of solutions. In each iteration the poor solutions are eliminated and the best-fit solutions are selected as parents. These are then 'recombined' to generate the individuals of the population for the next iteration. The fitness of each solution is calculated using a fitness function or some other kind of quality measure.

Genetic Algorithms

Genetic Algorithms (GAs) belong to the family of Evolutionary Algorithms, which mimic the process of natural evolution. A population of strings in the search space, referred to as 'chromosomes' or the 'genotype' of the genome, are used to represent a set of candidate solutions referred to as 'individuals' or 'phenotypes'. A *genome* is made up of primary data types known as *genes* which are used to model the behaviour of natural genomes as they evolve.

Chromosomes can be represented as bit string, real numbers or lists of rules. A standard representation for a GA-based solution is usually a set of binary strings (from the binary alphabet $\{0, 1\}$) that can uniquely be mapped onto the chromosome structure. Real-valued encodings are often used in the context of test data generation, which is thoroughly described in Section 2.4.

To solve a problem, a GA-based solution initially generates a population of random solutions known as chromosomes or genomes. In each generation, the fitness of every individual in the population is evaluated, and multiple individuals are stochastically selected from the current population (based on their fitness). The selected genomes are then recombined to form a

new population (see Figure 2.6). This iteratively continues until either the maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. The process consists of *selection* [47], *crossover* [62], *mutation* [59] and *reinsertion* [121] operators as described below:

Selection – This is the first phase of a GA, in which a set of individual genomes are selected from the current population to recombine as parents for the next generation. The selection is performed based on the fitness of each individual, and is assessed using various selection algorithms including *fitness proportionate selection* [59] and *stochastic universal sampling* [21].

In fitness proportionate selection, also known as roulette wheel selection, the fitness of each candidate solutions is used to assign a probability selection score for that solution. The selection probability of an individual with the fitness of f_i would be $\frac{f_i}{\sum_{i=1}^{N} f_i}$, where N is the number of individuals in the population [59].

Stochastic universal sampling (SUS) is the developed version of the fitness proportionate selection (FPS). In FPS 'several' solutions from the population are used by repeated random sampling. In SUS, however, a 'single' random value is used to sample all the solutions by choosing them at smoothly spaced intervals. This provides the weaker individuals in the population a chance to be selected and therefore reduces the bias in the fitness-proportional selection methods. For more information about this and other selection algorithms, the reader is referred to references [21, 47, 75].

Crossover – During this phase a new population of offsprings is generated from combining the selected individuals (parents) from the previous stage. Crossover is also referred to as 'reproduction' or 'recombination', and is a critical feature of Genetic Algorithms. There are various methods for crossover including multi-point crossover [62] and uniform crossover [61].

In multi-point crossover [62], n crossover points are selected at random without duplications, which are arranged into ascending order. The variables between successive crossover positions are then exchanged between the two parents to produce two new children.

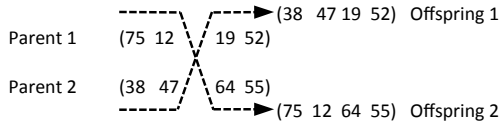


Figure 2.7: One-point crossover (recombination) in GA

The simplest variation of multi-point crossover is one-point crossover, in which the same crossover point is selected in both parent, and the variables on the right or the left of the crossover point are swapped to produce two new offsprings that embody the characteristics of their parents. This process is demonstrated in Figure 2.7. In uniform crossover [61], every position in the chromosome is a potential crossover point.

Mutation – After recombination every offspring (chromosome) undergo *mutation*. Offspring variables are mutated by small perturbations with low probability in order to preserve and introduce genetic diversity into each chromosome from one generation to the next. Genetic diversity is required to prevent the population of chromosomes from becoming too similar.

Genetic diversity for binary encodings is achieved by flipping bits of the binary strings at low probability p_m (i.e. usually less than 0.01). For real-valued encodings, a gene in the chromosome is replaced with a new value generated at uniform random or using the gaussian mutation [111], where the new value is selected using a gaussian distribution around the current value. In this thesis, the uniform mutation is used to mutate the individuals in the course of test data generation

Reinsertion – This is the final phase of a GA. During this process, the offsprings are reinserted into the old population if the number of offspring produced are more or less than the size of the original population. This is a necessary step to maintain the size of the original population, and to determine which individuals are to exist in the new population.

There are various schemes for local and global reinsertion [99, 121]. In local selection, individuals are selected in exactly the same neighborhood. Therefore, the locality of the information is preserved. The global reinsertion schemes include *pure reinsertion*, *uniform reinsertion*, *elitist reinsertion*, and

fitness-based reinsertion.

Pure reinsertion produces as many offspring as parents and replaces all parents with the offspring. This is the simplest reinsertion scheme, in which, each individual persists only one generation. However, it is very likely, that very good individuals are replaced without producing better offspring. *Uniform reinsertion* produces less offspring than parents and replace parents uniformly at random. *Fitness-based reinsertion* produces more offspring than required for reinsertion and reinserts only the best offspring. An *elitist strateg* to reinsertion replaces the worst of the current generation with the best offspring.

2.4 Search-Based Test Data Generation

The application of search-based techniques on test data generation requires defining and converting testing criteria into a set of objective functions that can be resolved using an appropriate meta-heuristic technique. These procedures can be summarised into the following steps:

Specifying a Testing Criterion – A common criterion for structural testing is branch coverage. Based on this objective, locating a set of test inputs that can obtain the desired level of branch coverage is required.

Representation of Candidate Solutions – The candidate solutions for the problem at hand must be capable of being encoded so that they can be manipulated by the search algorithm. This representation is usually sequences of elements (e.g. binary, or real values) which form individuals such as chromosomes in a GA. In test data generation, the most common representation of candidate solutions is real-valued encodings, where the input vector to the program is the direct representation of a candidate solution.

Outlining The Program's Input Domain – The program's input domain and the search space must be identified. The search space is usually formed from a combination of all the possible values in the program's input domain.

Table 2.2: Tracey's distance functions [115, 116] make use of a non-zero positive constant K, which is always added if the term is not true. This allows the objective function to always return a non-zero positive value when the predicate is false, and zero when it is true.

Boolean	if TRUE then 0 else K
a > b	if (b - a) < 0 then 0, else(b - a) + K
a ≥ b	if (b - a) ≤ 0 then 0, else(b - a) + K
a < b	if (a - b) < 0 then 0, else (a - b) + K
a ≤ b	if (a - b) ≤ 0 then 0, else(a - b) + K
a = b	if abs(a - b) = 0 then 0, else abs(a - b) + K
a ≠ b	if abs(a - b) ≠ 0 then 0, else K

Defining the Fitness Function – A fitness function must be formulated to computes the fitness of the candidate solutions with respect to a set of objective functions. The description of the fitness function used in this thesis is presented in Section 2.4.1.

2.4.1 Fitness Function for Branch Coverage

The search-based test data generation is essentially the process of reformulating a testing criterion into an objective function, and determining a *fitness function* that can guide the search towards appropriate test data with respect to the defined testing goal.

The common testing goal in structural testing is branch coverage. The objective function designed for branch coverage must compute the fitness of candidate test inputs in terms of their branch-covering criterion. This involves assessing how far away the candidate solutions (test inputs) are from executing the target branch. This objective function consists of two major components *approach level (AL)* [119] and *branch distance (BD)* [20, 54, 79].

The approach level is an integer value indicating how close a candidate solution is to a target node in terms of the program's CFG. This is achieved by counting the number of unexecuted nodes on which the target node is transitively control dependant. The branch distance indicates how close a predicate is to being true, and its calculation varies depending on the corresponding relational predicates. Table 2.2 shows a list of branch distance functions for various relational predicates defined by Tracey [115].

The use of approach level and branch distance is demonstrated as an example shown in Figure 2.8. To produce a test input that causes the execution of the true branch of the predicate $if(p1value < 50)$ (in line 15), the branch distance is defined as 0 if $p1value - 50 < 0$, or otherwise $p1value - 50 + K$. The value K in this case is a positive constant which is always added if the term is not true. This information is continuously updated as to guide the search until the test data of interest is discovered or resources exhausted. The 'closer' the output of the branch distance is to zero, the 'closer' the search-based technique is to finding the test data of interest.

Formally, the fitness function used for search-based test data generation is computed as follows [20]:

$$fitness = AL + normalise(BD) \tag{2.1}$$

where the branch distance BD is normalised into the range [0, 1] using the following function [20, 120]:

$$normalise(BD) = 1 - 1.001^{-BD} \tag{2.2}$$

This formula ensures the value added to the approach level is close to 1 when the branch distance is very large, and 0 when the branch distance is zero.

2.4.2 IGUANA

IGUANA [80] is a search-based test data generator tool for C programs. It has an object-oriented architecture and is written in Java. The meta-heuristic strategy employed in IGUANA initially generates a random input vector for a test object and then modifies the input based on the information obtained from a fitness function. In this process, the search space is formed from a set of possible input vector parameter-value combinations. The test object is instrumented to return fitness information. The search applies this information to explore promising areas of the program's input domain.

```
1  public boolean isValid(String casNumber) {
2    boolean overall = true;
3
4    //check format
5
6    String format = ``^(\\d+)-(\\d\\d)-(\\d)$'';
7    Pattern pattern = Pattern.compile(format);
8    Matcher matcher = pattern.matcher(casNumber);
9    overall = overall && matcher.matches();
10   if (matcher.matches()) {
11
12    //check number
13
14    String part1 = matcher.group(1);
15    String part2 = matcher.group(2);
16    String part3 = matcher.group(3);
17    int part1value = Integer.parseInt(part1);
18
19    // CAS numbers start at 50-00-0
20
21    if (part1value < 50) {
22     overall = false;
23
24    } else {
25     int digit = CASNumber.calculateCheckDigit(part1, part2);
26     overall = overall && (digit == Integer.parseInt(part3));
27    }
28   }
29   return overall;
30 }
```

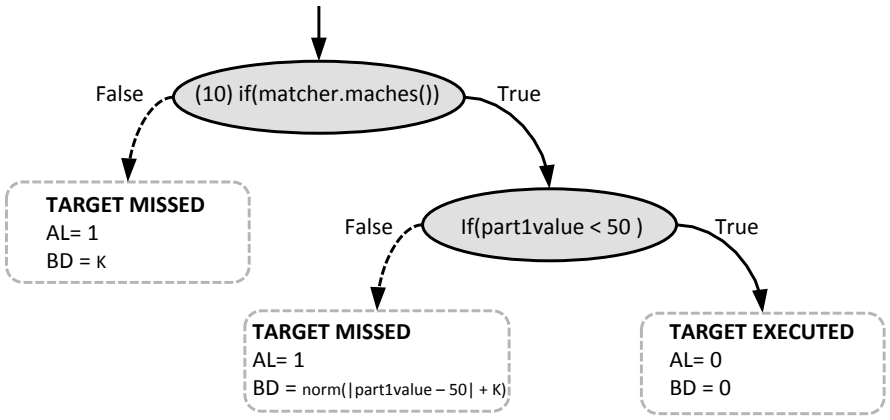


Figure 2.8: Computation of approach level (AL) and branch distance (BD) in test data generation for *isValid* method. AL represents the number of unexecuted nodes from the target node in the program's CFG. DB indicates how close a predicate is to being true. The function *norm* is used to normalise the DB in the range [0 ,1].

This framework can be used to investigate various search-based methods with different fitness functions, and program analysis techniques for test data generation.

2.4.3 Applying Alternating Variable Method

One of the earliest algorithms used in search-based test data generation is the *Alternating Variable Method (AVM)*, proposed by Korel [71]. The AVM firstly initiates all the input variables of the PUT with a random value, then attempts to improve the value of each variable in sequence. The method performs in two phases: the *exploratory phase* and the *pattern phase*.

In the exploratory phase the neighbourhood of the first variable is investigated by applying slight modifications to the initial value generated at random. This includes adding or subtracting a delta (δ) from the original value. In this phase, the delta is $dir \times 10^{-prec_i}$, where dir is either -1 or 1 representing the growth direction, and $prec_i$ indicates the precision of the i^{th} input variable. The precision only applies to floating point variables and is 0 for integral types. For example, setting the precision ($prec_i$) of an input to 1 restricts the smallest possible movement to ± 0.1. Increasing the precision to 2 limits the smallest possible movement to ± 0.01 [53].

If any of these alternations during the exploratory phase leads to a better solution s', it replaces the current solution (s) and the search enters the pattern phase. Otherwise the exploratory search is considered as unsuccessful and proceeds to perform exploratory searches on the remaining variables. This process continues until either a better neighbour is discovered or all the variables are unsuccessfully explored. In the latter case, the search restarts with a new random point. A similar procedure is applied for string types since strings are treated as vectors of integers representing the corresponding ASCII characters. In this sense, an initial solution for a string input is a vector of random integers [53].

In the pattern phase, the search attempts to accelerate towards an optimum by enlarging the size of the neighbourhood movements on every iteration. This involves making larger modifications to the current solution successively until an optimum is discovered. The amount of delta added or

subtracted from the current value during this phase is calculated using the formula: $\delta = 2^{it} \times dir \times 10^{-prec_i}$, where "*it*" represents the iteration index in current movement (pattern phase), and is used as a scale factor for the size of the neighbourhood movements [53, 54].

A series of similar movements is made until a minimum for the objective function is found for the input variable. Once no further improvements can be found for the input, the search continues optimising the next input parameter, and may recommence with the first input if necessary. In case the search stagnates, i.e. no move leads to an improvement, the search reinitiates at another randomly chosen location in the search space. This is referred to as a *random restart strategy* and is designed to overcome local optima by enabling the AVM to explore a wider region of the input domain for the program under test.

2.4.4 Applying (1+1) Evolutionary Algorithm

A (1 + 1) Evolutionary Algorithm is the most simple variant of an Evolutionary Algorithm which performs on population size of 1 [119, 118]. This simple EA merely operates based on selection and mutation due to the single-individual population. The term 'individual' refers to a 'search point', which is initialised with a randomly selected value using the uniform distribution. The value is then modified through random changes called mutations, and replaces the current one if it obtains a superior fitness [35, 72].

In the initialisation phase a value $x \in R^n$ is selected randomly using the uniform distribution. Next, the value x is selected as the current string in the selection phase. The mutation phase involving binary encodings (bit string representation x) requires flipping each bit x_i independently with mutation probability p_m.

The mutation for real-valued representations involves replacement of a gene (x_i) with a new value generated randomly using a gaussian or uniform distribution. The main scheme employed in this thesis is the uniform mutation, in which, the value of the chosen gene is replaced with a uniform random value selected between the user-specified upper and lower bounds for that gene.

```
Alternating_Variable_Method(all_variables: vector)
for (i = 1 → all_variables_size) do
    s ← random_solution ;
    if (obj(s_i) > 0) then
        j ← current loop iteration;
        s' ← exploratory_move(j, s_i);
        successful ← false;
        while (obj(s') > obj(s_i) && j < max_evaluations) do
            s' ← exploratory_move(j, s_i);
        end
        if (obj(s') > obj(s_i)) then
            successful ← true;
        end
        if (successful) then
            s_i ← s';
            k ← current loop iteration;
            while (obj(s_i) > 0 && k < max_evaluations) do
                s' ← pattern_move(k, s_i);
            end
            s_i ← s';
        else
            next variable;
        end
    end
    return s_i;
end
exploratory_move(s: candidate_solution, it: iteration)
dir ← ±1;
δ ← dir × 10^−prec;
s' ← s + δ;
if (obj(s') < obj(s)) then
    s ← s';
end
return s_i;
pattern_move(s: candidate_solution, it: iteration)
dir ← ±1;
δ ← ×2^it × dir × 10^−prec;
s' ← s + δ;
if (obj(s') < obj(s)) then
    s ← s';
end
return s_i;
```

Figure 2.9: Pseudo code illustrating the algorithm for Alternating Variable Method (AVM), *obj* is the objective function to be minimised. The search initiates with a random solution s which is modified though the exploratory and patternt phases (*exploratory_move* and *patternt_move*). δ is the amount added to or subtracted from the original solution during each phase. *dir* indicates the direction of the change (+1 or -1), and *prec* determines the precision of the input value.

```
Evolutionary_Algorithm()
s ← s ∈ S // choose an initial individual s uniformly from the
search space ;
repeat
    s ← mutate (s) // replace the gene with a new randomly
    generated value ;
    if (obj(s′) < obj(s)) then
        s ← s′;
    end
until (termination codition);
return s;
```

Figure 2.10: Pseudo code illustrating the algorithm for (1+1) Evolutionary Algorithm, with an initial solution s, and the objective function obj to be minimised. The termination condition is satisfied when the search executes the maximum number of fitness evaluations.

In the final phase the current string s is replaced with the new string s' if $f(x') \leq f(x)$, otherwise the current string s remains the same. The last two steps continue iteratively until the stopping criterion is satisfied.

The (1 + 1) EA is commonly regarded as a special variant of Hill Climbing and is referred to as a randomised or stochastic Hill Climber [12]. The (1 + 1) EA commences with one current point in the search space and always rejects a new point with inferior fitness value. However, as opposed to Hill Climbing, the search radius is not limited to the current point's neighbourhood. It can therefore reach to any point in the search space in one single step. Figure 2.10 presents a high level pseudo code for this algorithm.

2.4.5 Search-Based Test Data Reduction Techniques

Test data generation is the process of producing a set of test cases for the PUT under a pre-defined set of testing requirements. Test data generation for branch coverage is the process of producing a set of test cases, referred to as a *test suite*, which can cover all the feasible branches in the PUT. These test suites often contain a large number of test cases that must be executed and evaluated in the later phases. This is a time-consuming process that may have a severe impact on the overall costs of software testing.

Test data reduction techniques attempt to reduce the size of automatically generated test suites by identifying and eliminating the redundant and obsolete test cases and forming an optimal representative subset that can still satisfy all the pre-defined testing objectives. The majority of these techniques make use of source-code analysers and instrumentations to reduce the number of test cases in a given test suite while keeping test data adequacy unchanged. Despite the significant costs and effort preserved by these techniques, they are likely to decrease the fault-finding capabilities of the test suites [102]. Previous empirical studies provide conflicting evidence on this issue.

Wong et al [123] performed an empirical study to examine the effect on fault detection when reducing the size of a test set while keeping the coverage constant. A large collection of test cases was generated for a number of programs for branch coverage. The optimal subset of these test cases were then identified for each program by removing the redundancy while keeping the coverage constant. The authors reported that the fault-finding capabilities of both sets of test cases for each program remained unchanged. It was therefore concluded that test cases that do not add coverage to the test set are likely to be ineffective in detecting additional faults.

Rothermel et al [102] applied test suite reduction on a large number of test suites generated for various C programs. They assessed the size of the resultant test suites and their corresponding fault-fining capabilities. In contrast to the previous studies, the authors concluded that the fault detection effectiveness of test suites could be severely compromised by test-suite reduction techniques.

2.4.6 Seeded Search-Based Techniques

Search-based test data generation for branch coverage often results in production of arbitrarily looking and difficult-to-read values that are dissimilar to the test inputs a human would normally generate. Seeded search-based approaches are proposed to circumvent this issue by incorporating additional knowledge into the search-based mechanism.

A simple seeded search-based approach is to ‘seed’ the initial phase of

the search process with samples of appropriate inputs [82]. The local search-based algorithms such as Hill Climbing require only one appropriate input value to seed the search process. Global search algorithms such as GAs however rely on several starting points, and thus require at least a few appropriate input values as the starting points.

Another seeded search-based approach, proposed by McMinn et al [82], is to influence the search process towards existing points in the input domain, yet still with the target of covering a branch. A GA-based approach attempts to bias the search operators such as crossover and mutation towards the existing points. The biased operators would then search around these points attempting to generate similar values.

Following sections describe various approaches to the seeded search-based test data generation.

Search-Based Augmentation Approach

Harman and Yoo [125] proposed a seeded search-based approach in which a meta-heuristic search algorithm is seeded with existing test cases to generate additional test data in the context of regression testing. The employed search-based algorithm in this approach (i.e. Hill Climbing) seeks for a test case that behaves in the same way as the original one but has a different value. This is achieved by making two major alterations to the standard Hill Climbing technique.

Firstly the search is initialised with a global optimum that corresponds to the existing test data as opposed to adapting from random restart. This enables escaping from the local optima. Secondly, the search examines the neighbouring solutions in a random order and moves to the first neighbouring solution with a higher fitness value as opposed to following the conventional strategies *first-ascent* and *steepest-ascent* in Hill Climbing.

Figure 2.11 demonstrates the pseudocode for the augmentation technique. The implementation of this algorithm is based on two criteria: the *neighbourhood* that determines the neighbouring solutions (near the existing test data), and the *search radius*, which determines how far from the existing test data the search should run (given a definition of near neighbours).

```
Augmentation Algorithm()
currentSol ← existingSol;
while (within_the_search_radious) do
    if (neighbours of currentSol contains qualifying
    newSol) then
        currentSol ← newSol;
    else
        break ;
    end
end
return currentSol;
```

Figure 2.11: Pseudo code illustrating the augmentation algorithm

Semi-Automated Search-Based Approach

Pavlov and Fraser [97] presented a seeded search-based test generation approach in a unit testing scenario for object-oriented software. In this approach, the tester is included in the test generation process to improve the current solution when the search stagnates, and the improvements are then seeded back into the search. In this process, firstly an initial population of test suites is generated randomly, which is successively evolved using a GA. When the search mechanism in the GA stagnates, an editor window is displayed to the user presenting the current best test suite together with information on the coverage of the class under test. The tester then applies appropriate modification to the test suite as required. Finally the result is inserted back into the population, allowing the GA to continue the search. The use of this approach however is subject to the tester's input, as it is a semi-automated approach.

Seeded Search-Based Test data Generation For Strings

Alshraideh et al [16] presented an approach to automatic search-based test data generation for programs with string predicates such as string equality, string ordering and regular expression matching. In this approach, a new type of search operator was introduced with the intention of skewing the search towards strings that occur as literals in the program under test.

McMinn et al [84, 105] presented a seeded search-based approach for generating test cases involving string inputs. This approach attempted to collate samples of string inputs from the Internet by extracting and reformulating program identifiers into web queries. The resultant webpages were then downloaded and their content were split into tokens, which were then used to augment and seed the search-based test data generation process. The authors performed an empirical study, concluding valid and well-formed string inputs can be obtained for programs using web queries, and that the use of this technique can improve the branch coverage of the PUT. The use of this approach however requires the program to have useful identifiers that can be reformulated into web search queries, otherwise pages containing suitable strings may not be found.

Alshahwan et al [15] proposed a seeded search-based approach in the context of testing web applications. They developed a search-based test data generator tool, named as *SWAT*, which collates input values from the constants in the PHP programs. The values are then used to seed the search-based test data generation mechanism. The tool was designed to firstly present the target HTML pages in a structured form so that the constant values for different input fields could easily be extracted by the tester. It then seeded the collected values into the search space when targeting their associated branches. The authors performed an empirical study on the approach, and reported that the efficiency and effectiveness of the seeded approach was significantly enhanced in comparison to traditional search-based techniques.

Seeding Strategies

Fraser et al [42] empirically evaluated various strategies to seeding search-based test data generation for object-oriented software. The main objectives of this investigation was to firstly inspect the impact of the seeded search-based test data generation on the obtained results, and secondly, to identify the best seeding strategies used. In this study the authors applied all the following seeding strategies to generate test cases for a number of case studies:

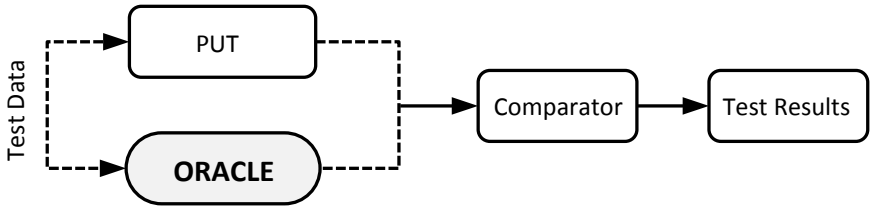


Figure 2.12: The test data evaluation process using an oracle

1. Using the constants such as numbers or strings extracted from the source code to seed the initial population of the search.

2. Applying pre-processing techniques that can improve the initial randomly generated population of the search in terms of diversity and suitability for the optimisation target.

3. Reusing the previous solutions such as existing or hand crafted test cases to seed the initial population of the search.

The authors concluded that the seeding strategies do improve the performance of the search-based test data generation. However, different strategies do provide different ranges of improvement. In some cases this effect depends on the type of the system under test. For instance, seeding strategies using constants from the source code can improve the performance, particularly for classes that rely on string objects. Seeding strategies using existing hand-written test cases can also improve the performance with high statistical confidence. Seeding strategies involving pre-processing techniques can improve the performance depending on the crossover operator and the employed pre-processing techniques.

2.5 Test Data Evaluation

Test data evaluation is the process of executing the PUT with a generated set of test cases and verifying whether it behaves as expected. This is an essential phase of software testing, which is subject to the existence of a system, commonly referred to as *oracle*, that can determine the expected outputs of the PUT. Figure 2.12 demonstrates the process of test data evaluation through the use of an oracle.

The term *oracle* was first used and defined by Howden [60] in 1978. Later in early 1980's, Weyuker [122] introduced the notion of non-testable programs, exploring various types of oracles. Non-testable programs were defined as those with complex computational patterns for which there exists no systematic oracles. Since then, various types of oracles have been proposed for different classes of programs. These include *pseudo oracles*, *specification-based oracle*, *heuristic oracle*, *statistical oracle*, *consistency oracle*, and *model-based oracle*. Each of these are discussed in following sections.

2.5.1 Pseudo Oracle

A pseudo oracle is defined as an independently implemented version of the PUT, which performs the same task as the original but using a different approach. A pseudo oracle can be implemented parallel to the original program using a different algorithm, a different programming language, or a different compiler so long as the original specification remains unchanged.

Davis and Weyuker [32] proposed the use of pseudo oracles for non-testable programs [122]. The idea was to run both program and its pseudo oracle on identical sets of test inputs, and compare the results. If the outputs revealed to be the same (or acceptably close in the case of numerical programs), the original program was considered to be validated. Otherwise, one of the programs were suspected to represent a failure. Examples of a pseudo oracle include programs that are implemented by different programming teams or in different programming languages. A pseudo oracle is expensive to implement, complex and often time consuming to run. Following section describes an approach to implementation of a pseudo oracle.

Testability Transformations

Automated generation of pseudo oracles was proposed by McMinn [83] through the use of *program transformations*. The idea was to transform an aspect of the PUT into an alternative version, which can be used as a pseudo oracle, and to compare outputs from the two versions so as to reveal potential failures in the original program. In the testing context, this type

of program transformation is often referred to as *testability transformations*, since it is mainly designed to improve the testability of a program either by improving test data generation or by acting as a pseudo oracle [28].

In [83], McMinn introduced the *Convert-to-BigDecimal* transformation as a pseudo oracle that can resolve issues regarding rounding errors in Java. Such errors usually occur when a program contains variables of primitive numerical types (e.g. int and long types in). Calculations on these variables normally leads to errors. Certain numbers of finite decimal representation (e.g. 0.1) is not supported in Java. For example, in Java, the operation 0.1+0.1+0.1 results in 0.30000000000000004 rather than simply 0.3. The accumulation of these errors can cause serious discrepancies during the course of the program.

To reveal such errors, the Convert-to-BigDecimal transformation uses the *BigDecimal* class in Java, and replaces variables of primitive numerical types with instances of the BigDecimal class. This transformation can also replace operations on these variables (e.g. $+, -, *$ and so on) with the appropriate method invocations (i.e. add, subtract, multiply etc.) on the BigDecimal object. Table 2.13 shows an example of a pseudo oracle implemented using Convert-to-BigDecimal transformation.

Testability transformation is a broad topic. As well as acting as a pseudo oracle [32], testability transformation has been applied to a wide range of testing scenarios to aid test data generation [49], improve code coverage and enhance program's semantics [81]. For more information about this see references [48, 83].

2.5.2 Specification-Based Oracle

Formal specifications are documentation methods that precisely declare the expected behaviour of the PUT through the use of mathematical notions. A popular example for these notations is the Z language [9], which has been used for writing formal specifications in various software programs. Other instances include the specification language of the Vienna Development Method (VDM) [40], and the Abstract Machine Notation (AMN) of the B-Method [11].

```
1 public void withdraw(long withdrawalAmount)
2 {
3   if (amount > withdrawalAmount)
4   {
5     amount -= withdrawalAmount;
6   }
7 }
```

```
1 public void withdraw(BigDecimal withdrawalAmount)
2 {
3   if (amount.compareTo(withdrawalAmount)>0)
4   {
5     amount = amount.subtract(withdrawalAmount);
6   }
7 }
```

Figure 2.13: Convert-to-BigDecimal Transformation on a simple function

For a program that is formally documented, its formal specification can be used to derive an oracle to determine whether or not the program performs correctly as expected. Formal verification techniques use this approach to verify the program's performance with respect to the existing specification [14]. The first approach was proposed by Richardson et al [101] in 1992. A model was developed to presents tools that enhance an integrated development environment, and enables the user to write formal specifications in a readable manner. The model then automatically derives test oracles from the generated specifications [95]. This approach consists of three essential stages; (1) write a complete specification of the required behaviour for the program in a formal notation, (2) generate test oracle from the specification, (3) run the program under test in the test framework using the test oracle to verify if it passes or fails.

Boyapat et al [22] developed a framework for automatically testing Java programs. For a formally documented method, the framework uses the method precondition to automatically generate a set of test cases. It then executes the method on each test case, and uses the method postcondition as a test oracle to check the correctness of each output. To generate test cases

for a method, the approach uses a class that represents the methods inputs. This class has one field for each parameter of the method and a predicate that uses the precondition to check the validity of methods inputs.

2.5.3 Invariant-Based Oracle

Ernst et al [36] introduced a technique, in which program's invariants (constraints) can be discovered using execution traces to help programmers identify program properties that must be preserved when modifying the source code. This technique is used to detect the explicitly stated invariants in a set of formally-specified programs, and also to infer likely invariants based on the values of variables in a program using a training test set.

Inferred invariants can be of substantial assistance in understanding, modifying, and testing a program that contains no explicitly-stated invariants. These can therefore be used as test oracle to check the correctness of a program's outputs. Daikon [37] is prototype a invariant detector tool that implements a set of techniques for discovering invariants from execution traces.

2.5.4 Metamorphic Based Oracle

Metamorphic testing is the process of generating additional test cases based on the existing test cases when the existing test cases do not reveal any failures. This is to allow a program to be further verified against some necessary properties, called "metamorphic relations". The main objective of Metamorphic Testing is to address part of the oracle problem.

Metamorphic relation is a property of a function that always persists among the multiple executions of the program under test. For example, a metamorphic relation $\sin(x + \pi) = \sin(x)$ is a metamorphic property of the function $\sin(x)$. This property can be applied on initial input x to produce $x + \pi$. This transformation allows the prediction of the output - $\sin(x + \pi)$, based on the (already known) value of $\sin(x)$. If the output is not as expected (if - $\sin(x + \pi)$ is not equal to $\sin(x)$), then a defect must exist. Performing metamorphic testing on a program involves:

1. Identifying metamorphic properties of a program.

2. Specifying the identified properties as a form of a formal specification.

3. Converting the specification into tests.

4. Running the test.

Metamorphic testing can be performed both on functional and system levels, Metamorphic testing at functional level focuses on individual functions rather than the application as a whole. This allows for the investigation of more metamorphic properties (and thus more test cases) and better fault localisation. Corduroy [90] is a tool which automates this process by allowing developers to specify individual functions' metamorphic properties using the specification language JML [92]. These properties then can be specified using an extension to JML, converted to test code, and then checked as the program runs on test input data.

Metamorphic testing at system level checks that metamorphic properties of the entire application hold after its execution. This approach treats the application as a black box and checks that the metamorphic properties of the entire application hold after its execution. Amsterdam [91] is a tool that allows checking of the application's metamorphic properties at runtime, using the real input from actual executions. Finding metamorphic properties of functions and applications is not always easy and straightforward. Furthermore, not all functions and applications have metamorphic properties.

2.5.5 Consistency Oracle

A consistency oracle refers to a simulator, an equivalent product, a software from a different platform, or a previous version of the PUT, which can be used to compare the results of one test execution with subsequent tests. The role of this oracle is to ensure the software is consistent in terms of the generated values and the end points.

The main application of a consistency oracle is Regression testing [93], where the key objective is to verify that alterations made to software have not adversely affected other parts. Various techniques have been reported

in the literature on how to select regression tests for program revalidation including [13, 27, 73, 77].

2.5.6 Heuristic Oracle

In situations where complete information is unavailable or impractical to acquire, a heuristic oracle promises to provide a significant process of verification. This type of oracle presents precise results for a few inputs and applies simpler consistency checks (heuristics) for the rest. This oracle essentially selects the known result for the exact comparisons and applies heuristic for the rest.

For example, heuristic oracle for the *cos*() function starts with the specific values for $\cos(\pi/2), \cos(\pi), \cos(3\pi/2), \cos(2\pi)$ (whose results are 0, -1, 0, 1). It then computes values between the four points at slight increments to the test object. A heuristic is applied to verify that the test object returns values that are progressively greater (or less) than the last value.

A heuristic oracle is relatively fast and efficient to create and to use. It can be useful when the program under test has a predictable relationship between the inputs and the outputs. For instance, a predictable relationship for the cos() function is that the function decreases between 0 and 180 degree and increase from 180 to 360. However using this approach, various parts of the PUT may remain unverified and thus systematic errors may remain undetected [57].

2.5.7 Human Oracle

In absence of an automated oracle, the human tester is required to manually interpret the generated test cases and the scenarios the represent by hand. In this process, the tester is expected to know the software operation and to identify the software failures. This process is highly depends on the complexity of the PUT, and is subject to the IQ skills of the tester who often requires books, tables, or calculators to determine the correct outputs of a program. This is an expensive, time consuming process that forms a significant cost, referred as the *human oracle cost*. The following section discusses various aspect of the human oracle in detail.

Test Data Quantity

One of the major issues associated with automatically testing is the overwhelming size of test suites, which has a direct impact on the costs and the effort of software testing. Literature of late has majorly focused on test data generation for branch coverage. This has led to production of large volumes of test data that requires corresponding amounts of oracle data for comparison.

Test suite reduction and test suite augmentation techniques promise to reduce this cost by minimising the size and enhancing the quality of generated test suites respectively [103, 125]. As discussed in Section 2.4.5, these techniques are however likely to affect the fault-finding capability of the resultant test suites.

Test Data Quality

In cerain instances the search-based approaches produce test cases that fall within one the following categories:

1. *Out of range numerical values* that are generated for primitive variables. These are digits that occur within $-32,768$ to $32,767$ range drawn from a 16-bit search space. For instance, a machine generated input for an integer type variable representing the calendar date is "*15148/26308/32447*", instead of a more readable value such as "*10/01/2013*".

2. *Non-alphabetical sequence of characters* that are generated for string variables. These are series of characters selected randomly from a search space of integers. The search space ranges from 0 to 127, which represents the printable characters defined by their ASCII codes. As an example, a machine generated string value for an email address is "*f#p%F@}UM%5.*6ZY*", instead of a more realistic value such as "*James@gmail.com*".

The quantitive aspect of oracle costs mainly refers to the difficulty of reading and comprehending machine-generated inputs, particularly string

values resembling arbitrary sequences of characters. Manual evaluation of such values and interpreting the scenarios these compromise is a difficult and time consuming task.

As previously described in Section 2.4.6, seeded search-based approaches can potentially circumvent the issues regarding the qualitative human oracle costs. The application of these techniques is however subject to availability of a permanent resource that can effectively incorporate some form of additional knowledge about the program's input domain into the search-based mechanism.

2.6 Mutation Analysis

The empirical assessment of test data generation techniques plays an important role in software testing research. In this thesis, *mutation analysis* is performed to assess the fault-finding effectiveness of the test suites generated using the proposed approaches.

Mutation testing was first introduced by DeMillo [33] as a fault-based testing technique, and was widely explored by Offutt et al [94]. The idea is to apply artificial faults (referred to as *mutations*) into the program producing *mutants*, and to execute both the program and its mutants with the same set of test cases. Mutants that result in different outputs to the original program are said to be *dead*, otherwise they are referred to as *live*. A live mutant indicates that the selected test set potentially fails to detect the introduced fault and therefore needs improvement.

The mutations are generated using *mutation operators* to represent typical programming errors. Mutation operators for imperative languages can be divided into two different categories *method level* and *class level*. The method level operators are used to modify an expression by replacing, deleting, and inserting primitive operators, and include:

1. Statement deletion.
2. Replace each boolean subexpression with true and false.
3. Replace each arithmetic operation with another one, e.g. $+$ with $*$, $-$ and $/$.

4. Replace each logical relation with another one, e.g. > with >=, == and <=.

5. Replace each variable with another variable declared in the same scope (variable types should be the same).

The class mutation operators are used to introduce syntactic alternations into the programs written in (Object-Oriented) OO languages [68] in order to produce mutants. The generated mutants can be classified into two types; *First Order Mutant (FOM)* which is generated by applying mutation operators only once. *Higher Order Mutant (HOM)*, which is generated by applying mutation operators more than once. Figure 2.14 demonstrate an example of each type of mutants.

While traditional mutation testing applies minor changes to the program's syntax, semantic mutation mutates the program's semantics (language) as opposed to its syntax. Semantic mutation aims to represent potential misunderstandings of the semantics of the language, and thus to capture a different class of faults. This type of mutation testing was first introduced by Clark et al [29], and has been implemented as a few tools including SMT-C for the C programming language [31]. For more information about semantic mutation operators the reader is referred to the references [29, 30, 31].

Mutation analysis is a method for assessing fault-finding capabilities of automatically generated tests suites. The main adequacy metric for this assessment is the so-called *mutation score (ms)*, which is defined as the percentage of mutants a test set T can detect (kill) over the total number of non-equivalent mutants. This can be calculated using the formula $ms(P,T) = 100 \times \frac{DM(P,T)}{M(P)-EM(P)}$, where $DM(P,T)$ is the number of mutants killed by the test set T, $M(P)$ is total number of mutants and $EM(P)$ is the number of mutants equivalent to P. A mutant is said to be *equivalent* if there exists no test case that can distinguish the output of the mutant from the output of the original program. The mutation score ranges from 0 to 100, where 100 is the best score possible, indicating that the specified test set can kill all the non-equivalent mutants. This score can be used to measure the effectiveness of a test set in terms of its ability to detect faults.

(a) First Order Mutant

```
1  boolean isValidProtocol(String protocol)
2  {
3    int len = protocol.length();
4    if (len > 1)
5      return false;
6    char c = protocol.charAt(0);
7    if (!Character.isLetter(c))
8      return false;
9    for (int i = 1; i < len; i++)
10   {
11    c = protocol.charAt(i);
12    if (!Character.isLetterOrDigit(c) &&
13      c != '.' && c != '+' && c != '-') {
14      return false;
15    }
16  }
17   return true;
18 }
```

(b) Higher Order Mutants

```
1  boolean isValidProtocol(String protocol)
2  {
3    int len = protocol.length();
4    if (len > 1)
5      return false;
6    char c = protocol.charAt(0);
7    if (!Character.isLetter(c))
8      return false;
9    for (int i = 1; i < len++; i++)
10   {
11    c = protocol.charAt(i);
12    if (!Character.isLetterOrDigit(c) &&
13      c != '.' && c != '+' && c == '-') {
14      return false;
15    }
16  }
17   return true;
18 }
```

Figure 2.14: (a) shows a First Order Mutant (FOM) for the program *isValidProtocol*, only one mutation operator has been applied to the original source code. This is shown in line 4, where the original operator $<$ is replaced with $>$. (b) shows a Higher Order Mutant (HOM), where more than one mutation operators have been applied to the original source code. This can be noticed from line 4, where the operators $<$ has been replaced with $>$, and the variable *len* have been modified to $len++$ in line 9.

Although mutation analysis was originally proposed as part of a testing strategy, it has been extensively used in literature as a method for evaluating various testing approaches. Andrews et al [18] performed mutation analysis to compare control flow and data flow test data generation techniques. Thevenod [110] used mutation analysis to evaluate test data generation using random and deterministic approaches. Bradbury [25] applied mutation analysis to compare traditional testing and model checking approaches. Other empirical studies in which mutation analysis is applied as an evaluation scheme include [17, 26, 68, 86].

In all these empirical evaluations, the researchers follow the pattern of generating a large test pool of test cases, executing the mutants with all the test data, and observing which test cases detect which faults. They then use this result to deduce the fault detection abilities of given test suites drawn from the pool.

2.7 Software Engineering Empirical Studies

Empirical studies play a fundamental role in science, supporting researchers gain knowledge by the means of direct observation or experience. Empirical studies in software engineering [10] often involve the scientific use of quantitative and qualitative data to understand and improve the software products or software techniques. The empirical data is essentially be obtained through the use of formal experiments, case studies, surveys, or prototyping exercises depending on the relevant research. A reliable empirical analysis compromise of the following essential components [98]:

1. Research context
2. Hypotheses
3. Experimental design
4. Threats to validity
5. Data analysis and presentation
6. Results and conclusions

The first step of an empirical study is to define a problem, and describe the relevant terminologies and background information surrounding the problem. The proposed problem is then formulated into a hypothesis or a research question to investigate (proposing hypotheses). An appropriate experimental design should be planned to obtain the data required for investigating the hypotheses. The experimental design is a detailed plan for testing the predictions and that can vary depending on the hypotheses and aim of the study. The results of the experiment is then analysed to conclude whether the theory on which the hypotheses were based is valid or not.

2.7.1 Human Empirical Studies

This section summarises some of the human empirical studies performed in software engineering fields.

Empirical Studies in Microsoft Research (MS)

The researchers of Microsoft Research (MS) [6] have performed various empirical studies in software engineering fields and different areas of computer science in collaboration with academic and industry researchers. This includes the study presented in [100], in which MS researchers investigated the effects of teams coordination in development of large-scale software systems. Large-scale software development requires coordination within and between large engineering teams that are located in different buildings, or different campuses of company, even in different countries with dissimilar time zones. The researchers investigated a 3 year old software application team, consisting of 300 people, based in Redmond. The study aimed to determine how the team coordinated with three intra-organisations, distributed in different locations.

The researchers interviewed 26 team members and revealed that how communication, capacity, and co-operation interchange influences the success of software development projects. They reported that the distributed teams faced additional challenges due to time zone and cultural differences between the team members. The researchers concluded that the majority of issues impacting engineers were not directly technical (e.g. code and APIs

related errors), but rather related to co-ordination issues between the team members.

In another study [19], MS researchers attempted to identify the co-ordination activities performed during bug rectifying of software systems. The study aimed to identify and explain the life cycle of bugs and the procedures of fixing the failure. This study investigated such co-ordination activities involved in bug fixing on software professionals at Microsoft. The study aimed to analyse the history of a closed bug in the database during the life of a project. The requested each person about the history of the last bug that they resolved or helped in resolving. From the results obtained, the study concluded that the histories of even simple bugs seriously depends on social, organisational, and technical knowledge and cannot be solely extracted from the automatic analysis of software repositories.

Software Engineering Observatory Project (in Sheffield)

The software engineering observatory project [8] was a large scale empirical study, started in 2000, as a collaboration between the Department of Computer Science and the Institute of Work Psychology at the University of Sheffield.

The study aimed to understand the processes that form the performance of software engineering, and to identify how these processes can be combined with human knowledge and technical factors. This involved observing various software developers while they were applying particular methodologies on real industrial projects. The software developers were undergraduate and postgraduate students who worked on internal and external software projects as a part of their course.

The empirical study included assessing the benefits of Extreme Programming (XP) [58], evaluating the relative merits of software development methodologies in terms of both the technical aspects and the well-being of the developers, identifying the factors that would form excellent team-based software development [64, 65, 66], and investigating the relative importance of the methodology adopted by the teams in [112].

Some of their findings concluded that effective software managers should

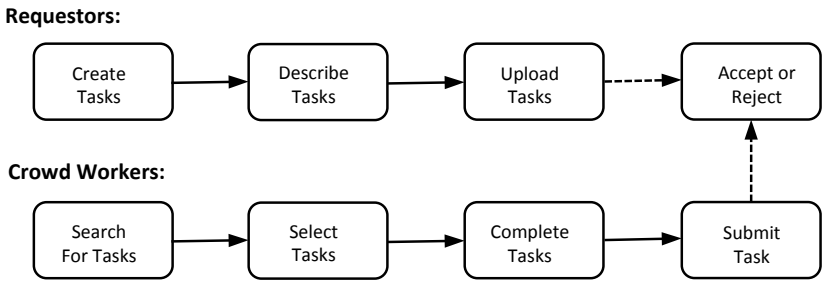


Figure 2.15: The crowdsourcing process

not only understand the technical aspects of the work undertaken by their staff, but should also understand their staff as individuals and how they can best work together in teams. It was concluded that teams without sufficient discussion on pertinent issues are most likely to encounter serious project problems. In addition, certain combinations of personality types that could be expected to be disruptive to the working of a software development team were also identified.

2.7.2 Crowd-Sourcing in Empirical Studies

Recruiting the right type and the right number of subjects is a challenging problem for empirical studies in software engineering. Researchers often use fewer participants of the right type to restrain to larger groups with the target population. An alternative solution is to use crowd-sourcing as a means to empirically assess a software engineering technique or a tool with aid of online users.

The use of crowd-sourcing enables a client referred to as *crowd-sourcer* to access a global community of users referred to as *crowd-workers* with different skills and backgrounds who can help performing a task. The task could be resolving a problem, classifying some data, refining a product or simply gathering some feedback.

There are various crowd sourcing platforms including Amazon Mechanical Turk (MT) [1], CrowdFlower [4], Crowd Guru [3], etc.

CrowdFlower the Crowd-Sourcing Website

CrowdFlower is one of the various platforms in which tasks or jobs can be uploaded for completion by crowd-workers for a fee. CrowdFlower divides complex projects into smaller manageable tasks that can be accomplished by a single person. These tasks usually require minimal time and effort, and users are paid a very small amount upon completion (normally a few cents). CrowdFlower prices each task based on the average time it takes the user to complete.

On the client side, crowd-sourcers order the number of crowd-workers they require for each task. The tasks are posted to a number of crowd-sourcing channels, which are then selected and accomplished by different contributors. Once the job is completed, CrowdFlower performs a quality control check, pays the crowd-workers, and then provides the data to the crowd-sourcers. The CrowdFlower's quality control check is performed using a set of hidden tests that are randomly distributed throughout the tasks and must be answered correctly.

Crowd-sourcing have been applied in a several software engineering empirical studies, including the investigating of code smells [109], fault localisation accuracy [44] and patch maintainability [43]. Crowd-sourcing has widely been used outside software engineering to support studies in human linguistic annotation [106] and Wikipedia article quality [70].

Kittur et al [70] explored the use of crowd-sourcing markets such as Amazons Mechanical Turk as promising platforms for conducting various human study tasks. The crowd-sourcing markets were reported as a suitable platform for recruiting a large number of crowd-workers to accomplish interactive tasks at marginal costs within a timeframe of days or even minutes. To maximise the capabilities of the approach, special care must be taken in the formulating subjective or qualitative tasks such as user measurements.

Snow et al [106] explored the use of Amazons Mechanical Turk as a significantly cheap and fast method for collating annotations from a broad base of non-expert crowd-workers over the Web. The authors investigated five different tasks including affect recognition, word similarity, recognising

textual entailment, event temporal ordering, and word sense disambiguation. The results revealed that annotations collated from Mechanical Turk non-experts highly matched with the existing gold standard annotations from experts.

Stolee et al [109] explored the use of crowd-sourcing to support empirical studies in Software Engineering. The authors assessed the impact of coding practices such as code smells [41] on the users preference and understandability of web mashups [34]. This study investigated the benefits of a crowd-sourcing platform such as Mechanical Turk to access and manage a large pool of study participants. Several issues were identified with regards to the implementation of effective crowd-sourced studies. These included the additional controls required to recruit the qualified users and to enhance the quality of the responses.

Fry et al [44] conducted a human empirical study involving a fault localisation task. The study aimed to assess the accuracy of human subjects in locating various types of defects in a few Java programs. For this purpose, faults were manually injected into the source code and 65 participants were recruited from Amazon Mechanical Turk to locate these faults. The authors concluded that certain types of defects were harder for humans to locate accurately, and certain code contexts were also harder to debug than others regardless of the type of defect involved.

Fry et al [43] presented another human study involving 32 real-world defects and 40 distinct patches. In this study, over 150 human subjects were recruited from Amazon Mechanical Turk to perform tasks that demonstrated their understanding of the control flow, state, and maintainability aspects of both human-written and machine-generated code patches. The authors reported that machine-generated patches were slightly less maintainable than human-written ones, however machine patches that were augmented with synthesised human-readable documentation presented a reverse trend.

Pastore et al [96] investigated crowd-sourcing as a mean to aid the oracle task during test data evaluation. The oracle process is split into small threads and uploaded onto the crowd-sourcing website, where it can be accessed and completed by the crowd. In this study, the authors presented the crowd users with assertions included in a test case, and requested them

to evaluate whether assertions reflected the current behaviour of the program. If the crowd determined that an assertion mismatches the program's behaviour (according to the code specification), then a bug has been located. The authors reported that crowd users can be used to automate the oracle problem, although obtaining appropriate results from the crowd is a notoriously difficult task.

2.8 Conclusions

This chapter explored the literature in the field of search-based structural testing. The operation of various search-based techniques employed in automated test data generation were discussed. This included Hill Climbing, Simulated Annealing and Evolutionary Algorithms.

The chapter then discussed the major issues associated with search-based test data generation including the qualitative and quantitive aspects and how these matters can affect the overall testing costs. This was proceeded with a discussion about the oracle problem and the description of various oracles, investigating how different aspects of automatically generated test data can particularly impact the human oracle costs.

The chapter then described empirical studies in software engineering and presented an investigation into mutation analysis as two major evaluation methods employed in this thesis.

Chapter 3

An Investigation into a Seeded Search-Based Approach For Branch Coverage

3.1 Introduction

This chapter applies a seeded search-based strategy to the automatic generation of test inputs, with the aim of producing branch-covering readable test inputs that are easy for humans to comprehend. As discussed previously, seeded search-based strategies incorporate the search mechanism with some additional knowledge [42, 82, 125], that can provide guidance towards promising areas of the search space. This knowledge is often in the form of sample test cases that are used as seeds to commence the search process. These test cases can be collated from various resources including program's source code, specifications, code comments [82], Internet web pages [84, 105] or the programmer themself. In this research, the seeds are collated directly from human subjects as described in Section 2.4.

An empirical study was performed to assess the effectiveness and efficiency of the seeded search-based approach in generation of branch-covering and fault-revealing test inputs. The case studies used in this experiment

included 14 Java methods from open source projects. The results revealed that test inputs generated using the seeded search-based approach obtained significantly higher branch coverage for 4 case studies. The fault-finding capability was also found to be improved for 9 of the case studies. The key contribution of this chapter therefore is:

> *The results of the empirical study in which both the seeded and unseeded search-based approaches are compared for generating branch-covering test suites, revealing cases where the seeded approach improves branch coverage, efficiency, and fault-finding effectiveness.*

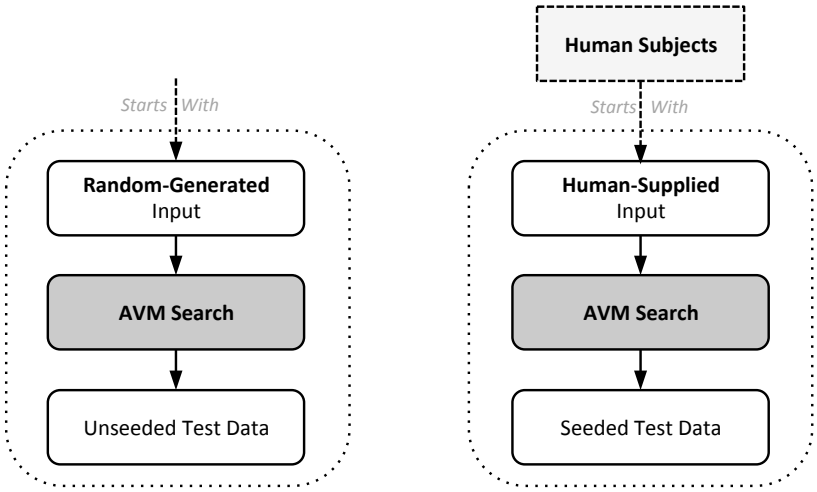
The chapter begins by describing the functioning of the search-based approach used in this study, and how it is seeded with additional information to guide the test data generation process. The chapter then describes the human empirical study, detailing the experimental setup. This is followed in Section 3.4, which discusses the statistical results on branch coverage and fault finding capability. Section 3.5 describes the threats to validity, while Section 3.6 concludes the chapter.

3.2 The Search-Based Technique

This study is focused on two variations of a search-based test data generation technique. The first approach applies the Alternating Variable Method (AVM) [71] to generate test inputs for a number of Java programs. This process commences with a randomly generated value in the search space, which is then continuously augmented during the *exploratory* and *pattern* phases (as detailed in the literature review Section 2.4.3). In this thesis, this approach is referred to as the *unseeded approach* in comparison to the *seeded approach* presented next.

The Seeded Approach

The AVM is configured to substitute the initial random value with a human-supplied value to guide the search procedure towards generation of similar test inputs. In this study, samples of test inputs are collated directly from



(a) Unseeded Approach (b) Seeded Approach

Figure 3.1: The operation of the seeded and unseeded search-based approaches. (a) demonstrates the basic performance of a standard AVM, where (b) represents the operation of a seeded AVM.

human subjects selected from two broad groups: students in the Department of Computer Science at the University of Sheffield, and Internet users participating via CrowdFlower [4] the crowd-sourcing website.

The human-supplied test inputs are predicted to enclose subtle knowledge about the program's input profile that is hidden to the current search-based heuristics. Seeding the AVM with these values should therefore result in generation of more branch-covering test inputs. The AVM is chosen as a local search method to ensure the final test data retains the readable aspects of the seeded inputs. Section 3.4 and Section 4.3 (of the next chapter) present the result of this investigation.

3.3 Experimental Study Methodology

An empirical study was performed to assess the branch coverage and fault-finding capabilities of the test data generated using both the seeded and unseeded approaches. This involved collecting samples of test cases from human subjects for the 14 Java programs. The human-supplied values were

Table 3.1: Case studies selected from 7 open source projects. These programs cover the both primitive and string input types.

Project	Class	Methods	# Branches	Parameter Type
Apache Commons http://commons.apache.org/math	org.apache.commons.math.util.MathUtils	factorial	2	Integer
		gcd	18	Integers
		binomialCoefficient	10	Integers
		compareTo	4	Doubles
Calendar	Calendar	days_between	32	Integers
Chemeval http://chemeval.sf.net	org.openscience.cdk.index.CASNumber	isValid	4	String
Daikon http://pag.csail.mit.edu/daikon	daikon.split.SplitterJavaSource	getClassName	2	String
		protectQuotations	4	String
OpenJDK http://openjdk.java.net	com.sun.jndi.dns.DnsName	isHostNameLabel	4	String
		compareLabels	8	Strings
	com.sun.jndi.toolkit.url.GenericURLContext	composeName	4	Strings
		isValidProtocol	8	String
		getURLPrefix	6	String
PuzzleBazar http://code.google.com/p/puzzlebazar	com.puzzlebazar.client.util.Validation	validateEmail	24	String

then used as seeds to initiate the automated search-based test data generation process. The empirical procedures consisted of four key steps:

1. The selection of the case studies as the basis for test data generation.
2. The human study protocol related to the information presented to the participants and the responses collated.
3. The selection of the human participants.
4. The incorporation of the human-supplied values into the search-based test data generation process.

The description of these steps are presented in the following sections.

3.3.1 Case Studies

The empirical study was concerned with search-based test data generation for branch coverage. It was therefore important to include case studies that have a relatively complex branching structure, where generation of branch covering test cases is a challenge. Another criteria for selecting case studies

for this experiment was that the operation of each case study is amenable to being understood from no more than a paragraph of text. The primary reason this was to avoid so-called fatigue effects, potentially biasing the results or increasing the number of unusable responses.

The case studies used in the experiment comprised of 14 Java methods with various primitive input types (string, integer and double) selected from 6 open source projects. The projects and methods are described in more detail as follows, with a summary presented in Table 3.1.

Apache Commons Mathematics is a library for mathematics and statistics. Four different methods were selected from this project. The method *factorial* computes the factorial of a non-negative integer n. The method *binomialCoefficient* computes the binomial coefficient $\binom{n}{k}$ which is the number of ways of selecting k unordered outcomes from n possibilities. The method *compareTo* compares two numbers given some amount of allowed error. It returns 0 if the two numbers (x and y) are equal, or -1 if x $<$ y, otherwise it returns 1. The method *gcd* computes the greatest common divisor of two integers.

Calendar is an open source program that computes the number of days between the two given dates. Method *days_between* was selected from this class.

Chemeval is a chemical evaluation framework for inspecting the molecular structures and the potential risk assessment. One method was selected from this project: *isValid* ensures that an input string is a valid CAS number. A valid CAS number is a string consisting of up to 10 digits (beginning at $50-0-0$), separated by hyphens, the last digit serving as a check digit.

Daikon is an invariant generator and a detector tool, used for reporting likely program invariants. Two methods were selected from its source code: *getClassName* deduces a Java class name from a string based on the final occurrence of the dot character. The method *protectQuotations* places a backslash in front of each quotation mark of a string.

OpenJDK is an open source implementation of the Java programming language, consisting of the Java Class Library and the Java compiler. Three methods were selected from this project. *isValidProtocol*, checks the validity of a protocol name. A valid protocol name is a string of which the length is

greater than one, the first character is an alphabetic letter (upper or lower case) and each of the remaining characters are either digits, alphabetical letters or any of these characters: {+, -, .}. The method *isHostNameLabel* takes a string as an argument, and returns true if the string is a valid host name. A valid hostname is considered to be a string where the first and the last characters are alphanumeric. The remaining characters may be alphanumeric or hyphens. The method *composeName* takes two strings *name* and *prefix* as arguments. If one of *name* or *prefix* are null or empty, the method returns the null or empty argument, otherwise it returns *prefix* appended by a forward slash followed by the *name*. The method *compareLabels* compares two labels alphabetically, ignoring case differences.

PuzzleBazar is a web-based system for creating, uploading and playing various puzzles including learning tools and tutorials. The method *validateEmail* selected from this platform checks whether the string argument is a valid email address.

The selected methods compromised of at least 2 to 32 branches (see *factorial* and *days_between* as examples). A number of these case studies had relatively complex, unstructured control flow and unbounded loops, such as *isValid*, *validateEmail*, *isValidProtocol* and *getURLPrefix*.

These case studies were categorised into one of the three classes *numerical computation*, *string validation* and *string conversion* routines. Numerical computation routines take one or more numerical inputs and return a numerical output, performing some form of computations on the inputs. These include *factorial*, *binomialCoefficient*, *compareTo*, *gcd*, and *days_between*.

String validation routines take one or more string inputs and return true or false, and include *isHostNameLabel*, *isValidProtocol*, *isValid*, and *validateEmail* methods. Conversion routines take one or more string inputs and return some string/integer output. These include *composeName*, *getClassName*, *protectQuotations*, and *compareLabels*.

3.3.2 Human Study Protocol

This phase of the empirical study was concerned with the experimental setup for collating samples of human-supplied test cases for the 14 Java methods

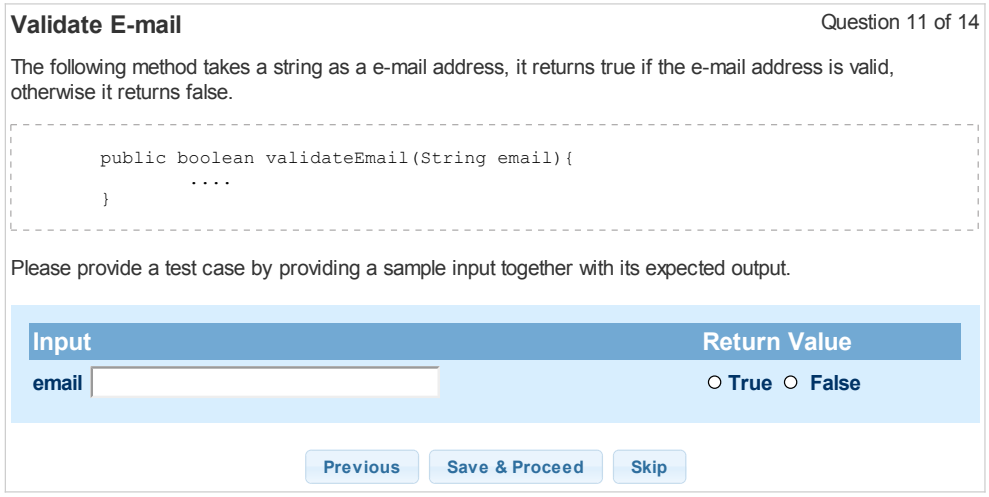


Figure 3.2: The online TCCollector application displaying one of the 14 questions

detailed in Table 3.1. This process was automated using a web application, referred to as *TCCollector*, which primarily presented a brief description about the study, and requested the participants to provide their level of education and their field of expertise. The application then presented a brief description about each of 14 Java methods sequentially, and requested the participant to supply a sample input and output for each method based on the description provided. The participant had the opportunity to either supply an answer, skip the question, or go back and edit their previous answers. Figure 3.2 shows a screenshot of this application with an example question.

3.3.3 Participant Selection

Participants for this study were selected from two groups: students from the Department of Computer Science at the University of Sheffield, and crowd-workers participating via CrowdFlower the crowd-sourcing website. All participants were required to have some level of self-reported experience in computer programming. Students were contacted via emails, and a total of 29 students participated in the study. One participant chosen at random was awarded with a 50 pound voucher token at the end of the study.

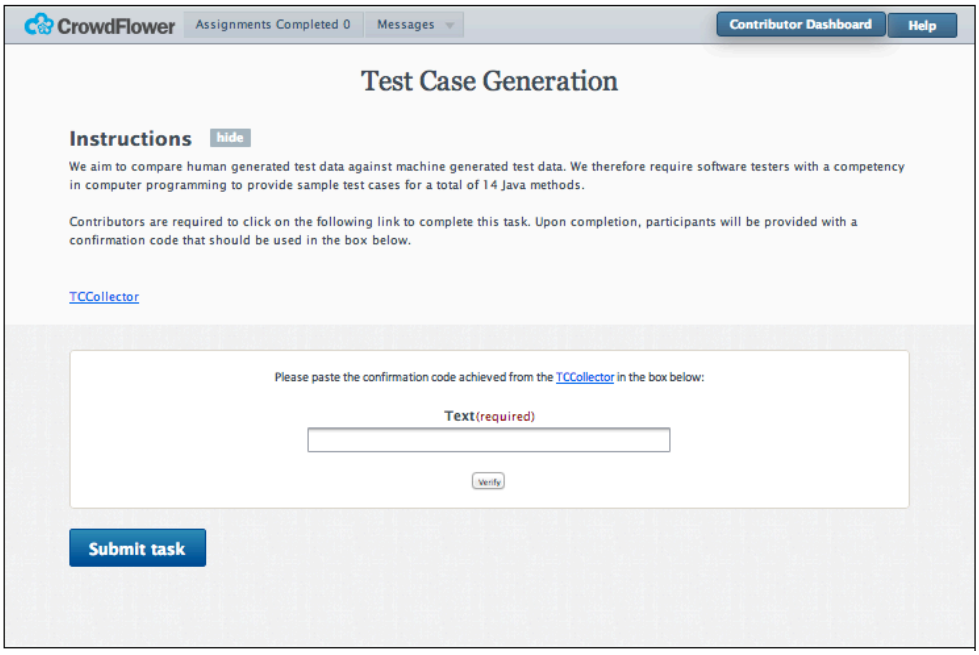


Figure 3.3: The task interface for the crowd-workers. A link to the TC-Collector is posted in the task's description panel. crowd-workers are then redirected to the application by clicking on the link. Upon the task's completion, the qualified participants are provided with a confirmation code, which should be pasted into the specified text field, in order to claim their payment.

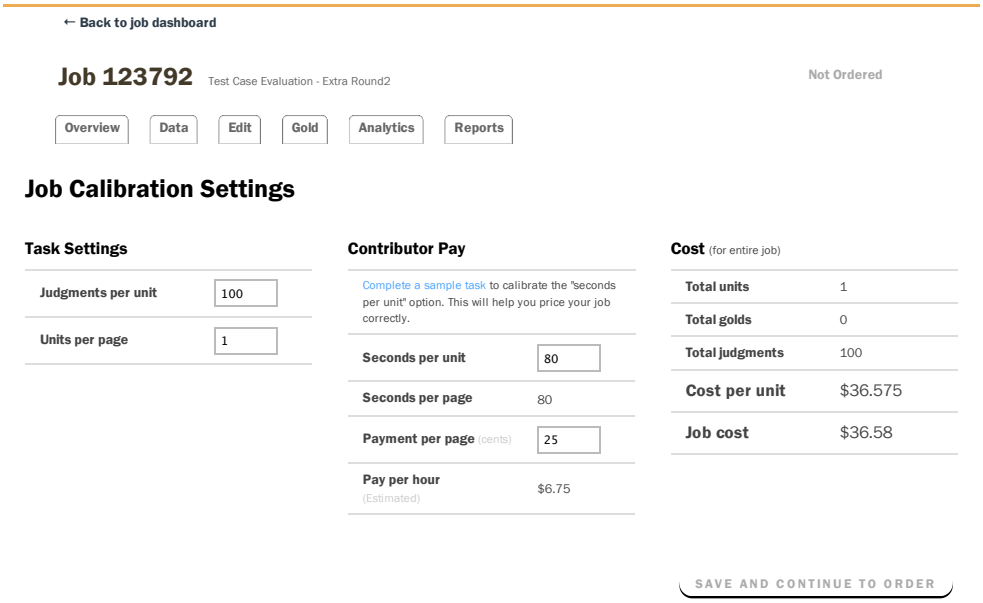


Figure 3.4: CrowdFlower Job Calibration Settings, where the requester specifies the job details and purchases the required number of judgments. The job then appears on a number of crowd sourcing channels, and crowd-workers can earn money by completing it.

Recruitment of crowd-workers was completed using a CrowdFlower account, where people can create and run tasks to be completed by crowd-workers for some fees. The number of crowd-workers required for this experiment, and its price was specified in the job's calibration settings panel. Once the order was purchased for the given price, the job would become available to the crowd-workers (or individual contributors) for completion.

Figure 3.3 shows how the job appears to the crowd-workers, while Figure 3.4 shows a screen shot of the calibration settings panel, indicating the purchase price paid for the 120 participants ordered for this experiment. The labour fee for each participant was specified as 25 cents, which would be given to them upon completion of the task. The total cost for this experiment was 36.58 USD (117×25) including the additional markup for the CrowdFlower labor costs.

CrowdFlower provides an opportunity for a wide range of people from all over the globe to accomplish tasks anonymously and earn money. While this allows a large number of participants to take part in a short time, it also increases the risk of participants trying to game the system for money by entering invalid responses. To resolve this issue, a number of crowd-sourced human studies have proposed to consider the responses from only a limited subset of participants based on some adequate metric [44, 43, 106, 70].

In this study, selection of participant from CrowdFlower was performed based on a *suspiciousness score*, which was assigned to each participant, and was set to 0 by default. This score would increment every time the participant entered some invalid data in the provided text fields for each question. The data validity was detected using different validation routines employed for each method. For example, string values or alphabetical characters entered as an input value for *factorial*, *gcd*, *days_between*, and *binomialCoefficient* would be considered as invalid, since these methods only operate on integer values. Participants who entered over 7 (out of 14) invalid responses were disqualified, and were subsequently removed from the study without payment. A total of 33 participants scored high suspicious scores and therefore were discarded from the total 150 crowd workers recruited for this study.

Data obtained from students were also checked for the suspiciousness

score using the same scheme. The results revealed a suspiciousness score of 0 in all cases, indicating that the data obtained from students did not contain any invalid responses, and thus, none of the student participants were disregarded from the study.

Identify Correct Test Case

A validation scheme was then performed to identify correct and unique test cases provided by human participants for each method. A correct test case was defined as a test case in which the input and output values were as expected given the method's description. A unique test case was defined as one which was never repeated in the obtained set of test cases for that method.

Table 3.2 shows the number of unique and valid test cases provided by both crowd-workers and students. These test cases were used as seeds to incorporate the test data generation process as explained in the next section.

3.3.4 Generating Test Inputs

This phase of the empirical study was concerned with generating test inputs for each method using both seeded and unseeded search-based approaches. Firstly, the standard AVM [71] was used as the unseeded search-based approach to attempt the full branch coverage of each method, within the maximum allowance of 1000 fitness evaluations involving each branch.

Due to the stochastic nature of meta-heuristic algorithms, the AVM was repeated 50 times on each method with different random seeds. This was to enlarge the test data's sample size, and to avoid the potential source of bias. The number of test suites generated using this approach was therefore 50 for each method.

Next, the full branch coverage of each method was attempted using the seeded search-based approach. In this stage, the AVM was seeded with a set of correct and unique test inputs supplied by human participants for each method. The search process for test data involving each branch was allowed up to 1000 fitness evaluations. The seeded AVM was repeated for each program a variable number of times depending on the number of correct

and unique test inputs obtained from the participants. Thus, the number of test suites generated using the seeded search-based approach differed for each case study. This is shown in Table 3.2

The search-based testing framework, IGUANA [80] , was used to perform the test data searches using each approach. In this study, IGUANA was easily adapted to Java as the selected methods were all static and hence there was no need to make a method call sequence. The representation for string inputs were defined as an array of integers representing a sequence of ASCII characters, s, with maximum length of 50, followed by an additional integer l, for controlling the string's length.

Using this representation, a sequence of 51 integers $(l_1, l_2, ..., l_{51})$ are generated, where the integer l_{51} is used to specify the length of the string input generated for the method. For instance, if the last integer of the sequence s is 8, the sub-sequence $(l_1, l_2, ..., l_8)$ is formed. This sequence corresponds to a sequence of ASCII character $(c_1, c_2, ..., c_8)$ representing the string input.

Each integer of the sequence $(l_1, l_2, ..., l_8)$ is derived from the ASCII printable range of 32 to 126. The test inputs for integer data types ranges from $-32,768$ to $32,767$ which is the range of the short type in Java.

3.3.5 Basic Definitions

Branch Coverage. Branch coverage is defined as the percentage of the program's branches a test suite can cover.

Mutation Score. Mutation score is defined as the percentage of mutants (faults) a set of test cases can detect (kill). More information about this was presented in Section 2.6.

Success Rate. Success rate is defined as the percentage of all the runs (i.e. 50) for which the test data to execute the branch is found. Success rate is a basis on which the effectiveness of the search can be compared for the branch using different approaches (i.e. seeded vs unseeded).

3.3.6 Research Questions

The research questions to be answered by the empirical study are as follows:

RQ 1. Quality of Human-Supplied Seeds. This research question compares the correctness of test cases obtained from crowd-workers and students. To answer this research question, the percentage of the correct test cases over the total number of test cases obtained from students and crowd-workers is computed and compared.

RQ 2. Test Data Branch Coverage. This research question determines whether the use of human-supplied seeds in the search-based test data generation process can significantly increase the branch coverage of generated test inputs. To answer this question, the branch coverage of test inputs generated using both seeded and unseeded search-based approaches is computed and compared for each program.

RQ 3. Test Data Generation Effectiveness and Efficiency. This research question inspects whether the incorporation of human-supplied test inputs into the search-based test data generation process can significantly improve the performance of the search function in finding the test data of interest. To answer this question, the efficiency and effectiveness of each approach is computed based on the following criteria:

(a) Effectiveness: The number of times each branch is successfully executed over the total number of search runs.

(b) Efficiency: The number of fitness evaluations performed by the search to cover a branch.

RQ 4. Test Data Mutation Score. This research question investigates whether the use of human-supplied seeds in search-based test data generation has any significant impact on the fault-finding capability of the generated test inputs. To answer this question, mutation analysis is performed to compute the percentage of mutants that the test inputs for each approach can detect.

3.4 Experimental Results

This section examines the outcome of the human empirical evaluation, assessing each research question.

RQ1. Quality of Human-Supplied Seeds

To answer this research question, the percentage of correct test cases supplied by crowd-workers and students were computed and compared. The results are demonstrated as a bar chart in Figure 3.5. The percentage of correct test cases obtained from students was considerably higher than those obtained from crowd-workers only for the *isValid* method. In other cases this was marginally improved approximately by $5 - 10\%$.

As previously mentioned, the test cases obtained from 117 crowd workers (out of 150) were used in the study. This allocated each method 117 test cases obtained from crowd-workers, and 29 test cases from students.

The Fisher exact test with confidence level set to 95% was performed to indicate the significance between correctness of the test cases obtained from crowd-workers and students. The result of this, displayed in Table 3.3, revealed that there was no significant differences between these two categories as none of the corresponding p-values were below 0.05. In addition, obtaining test cases from crowd-workers was significantly faster. A total of 150 subjects participated via CrowdFlower within 61 hours, while it took more than 3 weeks to gather only 29 volunteer students to complete the study.

Test cases obtained from both groups of participants had numerous repetitions for a few case studies. Table 3.2 shows the number of unique and correct unique test cases obtained from crowd-workers and students for each method. As evident from these results, the method *factorial* received the lowest number of unique test cases from both groups of participants.

Determining the output of the *factorial* method requires tedious mathematical calculations due to its arithmetical structure. Manual computation of this function can therefore be more difficult and error-prone for input values greater than 10. The majority of participants provided values less than 10, which are easier to compute.

Table 3.2: The number of correct and correct-unique test cases obtained from crowd-workers and students. A correct test case refers to the one in which the two input and output are as expected according to the method's description. A unique test case is defined as the one which is never repeated in the obtained set of test cases for the method.

(a) Crowd-sourced Seeds

Project	**Method**	**Correct**	**Unique**
Apache Commons	factorial	86	8
	gcd	76	40
	binomialCoefficient	60	14
	compareTo	48	45
Calendar	days_between	54	48
Chemeval	isValid	38	28
Daikon	getClassName	45	45
	protectQuotations	38	35
OpenJDK	isHostNameLabel	79	76
	compareLabels	49	40
	composeName	59	59
	isValidProtocol	74	68
	getURLPrefix	23	23
PuzzleBazar	validateEmail	100	100

(b) Student-supplied Seeds

Project	**Method**	**Correct**	**Unique**
Apache Commons	factorial	23	12
	gcd	22	20
	binomialCoefficient	18	12
	compareTo	14	14
Calendar	days_between	16	16
Chemeval	isValid	13	13
Daikon	getClassName	14	14
	protectQuotations	14	14
OpenJDK	isHostNameLabel	25	25
	compareLabels	19	18
	composeName	10	9
	isValidProtocol	21	21
	getURLPrefix	10	10
PuzzleBazar	validateEmail	27	27

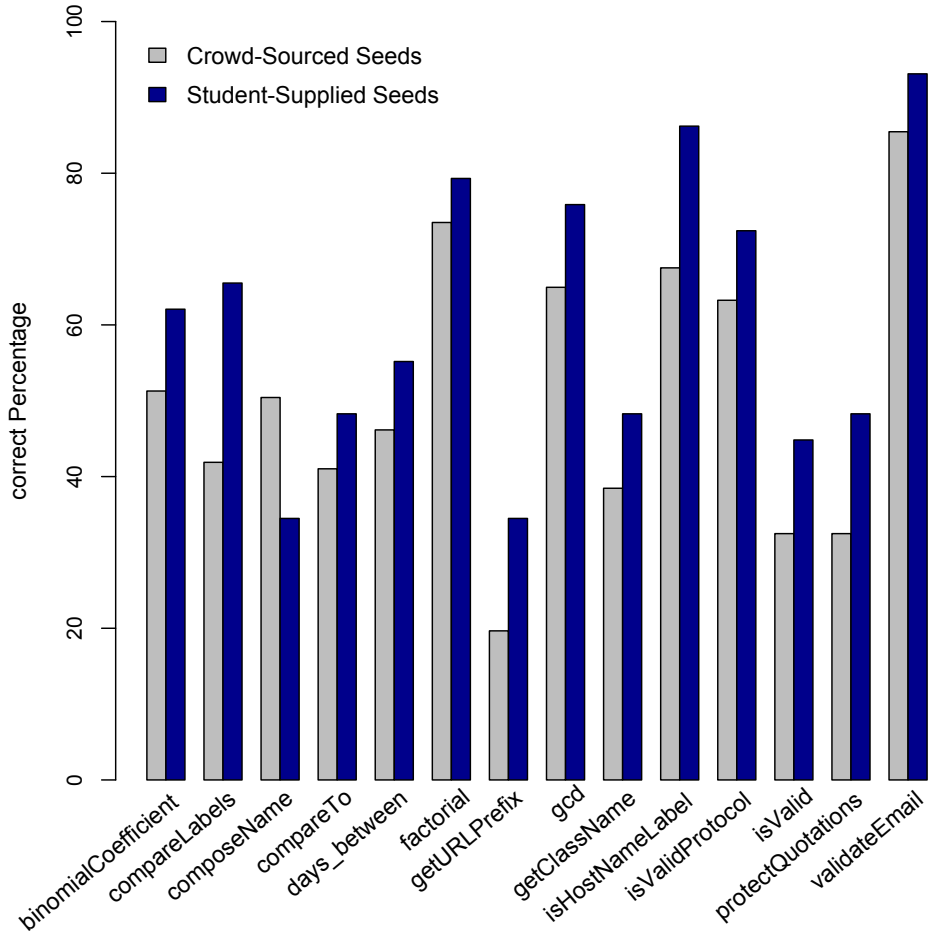


Figure 3.5: Percentage of the correct test cases obtained from crowd-workers and students

Table 3.3: The results of the Fisher test on correctness percentage of the test cases supplied by crowd-workers and students

Project	**Method**	**Correctness**		***p*-value**
		Crowd	**Student**	
Apache Commons	factorial	73.5	79.3	0.9
	gcd	65.0	75.9	0.6
	binomialCoefficient	51.3	62.1	0.6
	compareTo	41.0	48.3	0.7
Calendar	days_between	46.2	55.2	0.6
Chemeval	isValid	32.5	44.8	0.4
Daikon	getClassName	38.5	48.3	0.6
	protectQuotations	32.5	48.3	0.3
OpenJDK	isHostNameLabel	67.5	86.2	0.4
	compareLabels	41.9	65.5	0.2
	composeName	50.4	34.5	0.4
	isValidProtocol	63.2	72.4	0.7
	getURLPrefix	19.7	34.5	0.2
PuzzleBazar	validateEmail	85.5	93.1	0.9

RQ2. Test Data Branch Coverage

One of the key aspects of this study was to assess and compare the branch coverage of test inputs generated using the seeded and unseeded search-based approaches. Figure 3.6 shows the mean branch coverage for all the test suites generated using each approach for each program.

The Fisher exact test was performed with a confidence level set to 95%, to indicate cases where the seeded approach attained significantly higher branch coverage. Table 3.4 shows the results with significant p-values displayed in bold. As evident from these results, test cases generated using the seeded approach obtained significantly higher branch coverage for programs *getURLPrefix*, *validateEmail*, *isValidProtocol*, and *isValid* using the crowd-sourced seeds. The results from the students also followed a similar trend, with the exception of *isValidProtocol*. On average this improvement was 31.3% for the crowd-sourced data and 33.4% for the student data.

In answer to this research question therefore, the evidence suggests that the use of the seeded search-based approach has significant effects on the

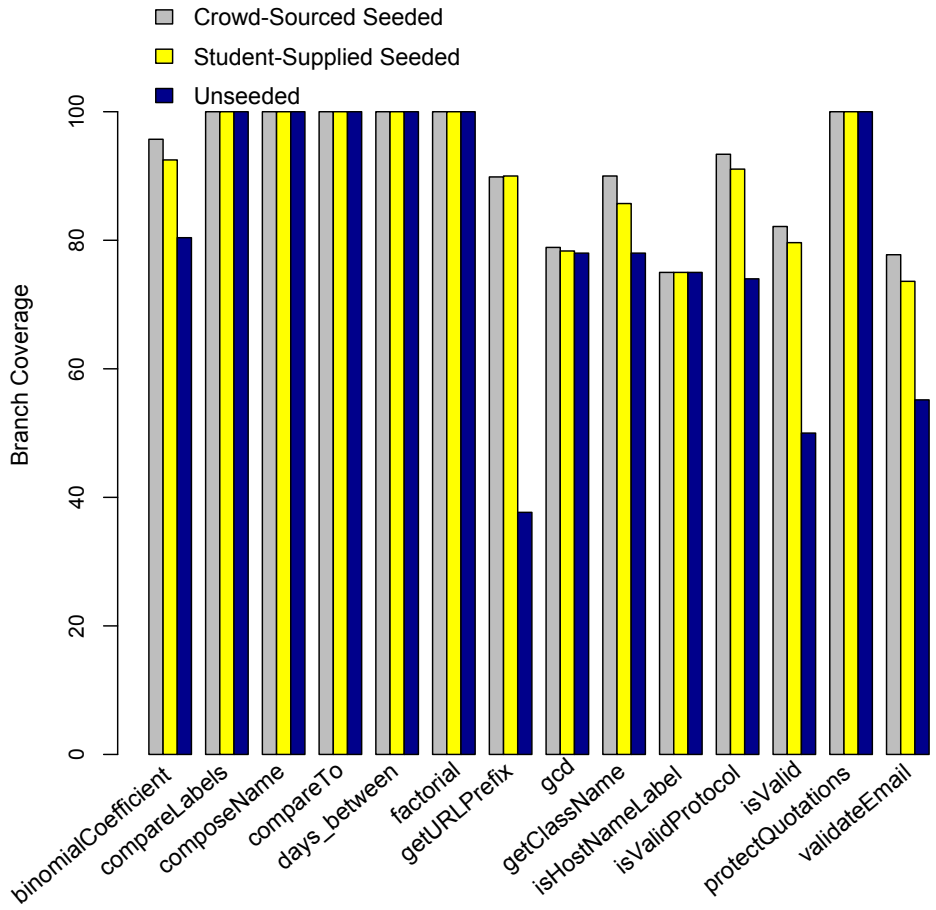


Figure 3.6: Effects of human-provided seeds on branch coverage

Table 3.4: The results obtained from the Fisher exact test on branch coverage of test inputs generated using both seeded and the unseeded approaches. (a) presents the crowd-sourced seeded test data, (b) presents the student-supplied seeded test data. Bold font indicates the significant cases (with p-values < 0.05) where the seeded approach outperforms the unseeded approach.

(a) Crowd-sourced Seeded Test Data

Project	**Method**	**Seeded**	**Unseeded**	***p*-value**
Apache Commons	factorial	100.0	100.0	1.0
	gcd	78.9	78.0	0.9
	binomialCoefficient	95.7	80.4	0.2
	compareTo	100.0	100.0	1.0
Calendar	days_between	100.0	100.0	1.0
Chemeval	**isValid**	**82.1**	**50.0**	**0.009**
Daikon	getClassName	90.0	78.0	0.5
	protectQuotations	100.0	100.0	1.0
OpenJDK	isHostNameLabel	75.0	75.0	1.0
	compareLabels	100.0	100.0	1.0
	composeName	100.0	100.0	1.0
	isValidProtocol	**93.4**	**74.0**	**0.019**
	getURLPrefix	**89.9**	**37.7**	**< 0.001**
PuzzleBazar	**validateEmail**	**77.8**	**55.2**	**< 0.001**

(b) Student-supplied Seeded Test Data

Project	**Method**	**Seeded**	**Unseeded**	***p*-value**
Apache Commons	factorial	100.0	100.0	1.0
	gcd	78.3	78.0	1.0
	binomialCoefficient	92.5	80.4	0.4
	compareTo	100.0	100.0	1.0
Calendar	days_between	100.0	100.0	1.0
Chemeval	**isValid**	**79.6**	**50.0**	**0.017**
Daikon	getClassName	85.7	78.0	0.9
	protectQuotations	100.0	100.0	1.0
OpenJDK	isHostNameLabel	75.0	75.0	1.0
	compareLabels	100.0	100.0	1.0
	composeName	100.0	100.0	1.0
	isValidProtocol	91.1	74.0	0.1
	getURLPrefix	**90.0**	**37.7**	**<0.001**
PuzzleBazar	**validateEmail**	**73.6**	**55.2**	**<0.001**

branch coverage of test inputs particularly for methods with string arguments. This is mainly due to incorporation of the human-supplied inputs that introduce additional guidance to the search process. There was no evidence to suggest that the use of the seeded search-based approach obstructs branch coverage in any of the cases, i.e. it did not reduce the branch coverage for any of the methods under consideration. This is in line with the results achieved by Fraser et al [42] as described in Section 2.4.6 of the literature review.

RQ3. Test Data Generation Effectiveness and Efficiency

A. Effectiveness

The success rate of the seeded and unseeded search-based approaches in executing individual branches were assessed using the Fisher's exact test. The termination criterion of each approach was set up to 1000 fitness evaluations for each branch. Table 3.5 and Table 3.6 present the significant results (p-values < 0.05) for the crowd-sourced seeded and student-supplied seeded test data respectively.

The Fisher test recorded a significant difference for 26 and 19 branches in each set of test data presented in Tables 3.5 and 3.6. In none of these instances, test data searches using the seeded search-based approach revealed to be detrimental. This implies that incorporation of the human-supplied test inputs (obtained from students and crowd-workers) into the search mechanism can improve the overall success rate of the test data generation process.

B. Efficiency

The performance efficiency of each approach was inspected based on the number of fitness evaluations performed in order to locate test data for each branch. The Wilcoxon rank sum test, with the confidence level set to 0.95%, was performed to check the statistical significance.

In addition, the Vargha and Delaney's $\hat{A}_{12}$ statistic [117] was used to assess the effect size. The $\hat{A}_{12}$ statistic computes the probability that a run of the first search-based approach executes a larger number of fitness eval-

Table 3.5: The significant results obtained from the Fisher's exact test on the success rate of the crowd-sourced seeded and unseeded search-based approaches (with the confidence level set to 95%). In all these cases the success rate of the crowd-sourced seeded approach is higher than the unseeded approach.

(a) Crowd-sourced Seeded Test Data

Method	**Branch**	**Success Rate**		***p*-value**
		Seeded	**Unseeded**	
binomialCoefficient	6T	64.3	2	< 0.001
	8T	92.9	4	< 0.001
isValid	4T	64.3	0	< 0.001
	4F	64.3	0	< 0.001
getClassName	2T	80	56	0.016
isValidProtocol	7F	100	70	< 0.001
	9T	100	70	< 0.001
	9F	100	70	< 0.001
	11T	100	70	< 0.001
	11F	95.6	60	< 0.001
getURLPrefix	4F	100	64	< 0.001
	7T	91.3	0	< 0.001
	7F	100	64	< 0.001
	10T	78.3	0	< 0.001
	10F	69.6	0	< 0.001
validateEmail	14F	86	50	< 0.001
	27T	86	52	< 0.001
	27F	86	36	< 0.001
	31T	86	38	< 0.001
	31F	86	52	< 0.001
	34T	81	50	< 0.001
	34F	86	52	< 0.001
	36T	18	6	0.049
	36F	81	50	< 0.001
	41T	86	52	< 0.001
	41F	86	38	< 0.001
	43T	86	36	< 0.001

Table 3.6: The significant results obtained from the Fisher's exact test on the success rate of student-supplied seeded and unseeded test data. In all these cases the crowd-sourced seeded approach has significantly higher success rate in comparison to the unseeded approach

(b) Student-supplied Seeded Test Data

Method	**Branch**	**Success Rate**		***p*-value**
		Seeded	**Unseeded**	
binomialCoefficient	6T	50	2	< 0.001
	8T	75	4	< 0.001
isValid	4T	59.3	0	< 0.001
	4F	59.3	0	< 0.001
isValidProtocol	7F	95.2	70	0.027
	9T	95.2	70	0.027
	9F	95.2	70	0.027
	11T	95.2	70	0.027
	11F	90.5	60	0.012
validateEmail	14F	85.2	50	0.003
	27T	85.2	52	0.006
	27F	77.8	36	< 0.001
	31T	77.8	38	0.002
	31F	85.2	52	0.006
	34F	85.2	52	0.006
	41T	85.2	52	0.006
	41F	77.8	38	0.002
	43T	77.8	36	< 0.001
	43F	51.9	2	< 0.001

uations compared to that of the second search-based approach. According to the guidelines presented in the Vargha and Delaney's paper [117], if the $\hat{A}_{12}$ <0.5, then the first search-based approach outperforms the second one, and the opposite is true if $\hat{A}_{12}$ >0.5. Also, depending whether the absolute difference $|\hat{A}_{12}$ - 0.5| is > 0.21, > 0.14, > 0.06, or $\leq$ 0.06, the corresponding effect size can be categorised as large, medium, small or negligible respectively.

Table 3.7 presents the significant results obtained from the Wilcoxon test and $\hat{A}_{12}$ statistic for both crowd-sourced seeded and unseeded test data. The Wilcoxon test recorded a significant difference in 75 branches. In 51 cases, the seeded approach performed significantly fewer fitness evaluations to cover the branch. However, for the remaining 24 branches, the seeded approach was detrimental, i.e. it performed more fitness evaluations to find the test data covering the branch.

Table 3.8 presents a similar result for the student-supplied seeded test data. The Wilcoxon test recorded a significant difference in 71 cases, in which the unseeded approach outperformed the seeded approach in 23 occasions. For the remaining 48 cases, the seeded approach performed significantly fewer fitness evaluations to cover the branch.

The branches for which the seeded search-based approach performed detrimental includes 4T and 11T in *binomialCoefficient*, 1F in *compareTo*, 6T, 32F, 39F, 43F, 47T, 47F, 49T, 49F in *days_between*, 2F in *getClassName*, 11T in *compareLabels*, 9F, 11T in *isValidProtocol*, 4T in *getURLPrefix*, and 20F, 34F in *validateEmail*.

The majority of these branches involve simple conditions for which the test data generation is a straightforward task. In such cases, the initial test input generated at random can often cover the target branch instantly. As a results, the search mechanism generates the required test data for that branch after performing the first fitness evaluation. The use of the seeded search-based approach can however divert the search towards different areas of the search space. In this case, the search mechanism would require to execute at least a few fitness evaluations in order to return to the required search point, and to locate the test data of interest.

In answer to this research question therefore, the evidence indicates that

Table 3.7: Results of the Wilcoxon rank-sum on the numbers of fitness evaluations completed by each approach to execute a branch. Columns **Seeded** and **Unseeded** presents the mean number of fitness evaluations for successful trials using the crowd-sourced seeded approach and the unseeded approach respectively. Values in bold face indicate the significant cases where the seeded approach outperforms the unseeded approach, italic font represents cases where the seeded approach performs detrimental. For the $\hat{A}_{12}$ statistic, * indicates a small effect size, ** a medium effect size and *** a large effect size, according to the guidelines of Vargha and Delaney [117].

(a) Crowd-sourced Seeded Test Data

Method	**Branch**	**FitnessEvals**		***p*-value**	$\hat{A}_{12}$
		Seeded	**Unseeded**		
factorial	2T	**3.4**	**7.4**	0.003	***0.202
	2F	*1.2*	*1*	0.014	*0.562
gcd	8T	**6.6**	**15**	< 0.001	***0.075
	8F	**2.4**	**31.8**	< 0.001	***0.262
	16T	*1.3*	*1*	0.005	*0.575
	19T	*1.2*	*1*	0.011	*0.562
	21T	*1.2*	*1*	0.011	*0.562
binomialCoefficient	2T	**3.1**	**12.1**	0.003	***0.24
	2F	**1**	**4.3**	0.046	*0.38
	4T	*562.4*	*158.3*	< 0.001	***0.907
	4F	**1**	**4.3**	0.046	*0.38
	8T	**2.5**	**380**	0.03	***0
	8F	*8.4*	*4.3*	0.031	**0.659
	11T	*45*	*24.6*	< 0.001	***0.794
compareTo	1T	**11.3**	**43.6**	< 0.001	***0.273
	1F	*13.4*	*6*	< 0.001	***0.724
	3F	**37.3**	**48.1**	< 0.001	***0.78
days_between	2T	**3.8**	**15**	< 0.001	***0
	4F	**1**	**11.5**	< 0.001	***0.25
	6T	*4.5*	*1*	< 0.001	***0.896
	6F	**1.8**	**15**	< 0.001	***0
	8F	**4.3**	**11**	0.002	**0.339
	10F	**1**	**9.5**	< 0.001	***0.25
	12F	**1**	**12.5**	< 0.001	***0.25
	14F	**1**	**8.9**	< 0.001	***0.25
	16T	**15.5**	**61.3**	0.037	*0.62
	16F	**1**	**12**	< 0.001	***0.25
	18F	**1.1**	**40**	< 0.001	**0.292
	28T	**4.9**	**120.6**	< 0.001	***0
	28F	*2.1*	*1*	< 0.001	**0.688
	32T	**71.3**	**206.3**	< 0.001	***0.168
	32F	*5.6*	*1*	< 0.001	***0.938

Continuation of Table 3.7 - Crowd-Sourced Seeded Test Data

Method	Branch	FitnessEvals		p-value	$\hat{A}_{12}$
		Seeded	Unseeded		
	34T	**72.4**	**206.3**	< 0.001	***0.168
	34F	**71.3**	**206.3**	< 0.001	***0.168
	39T	**38.3**	**73.8**	0.042	*0.382
	39F	*5.6*	*1*	< 0.001	***0.938
	43T	**6.6**	**31.9**	< 0.001	***0.764
	43F	*5.6*	*1*	< 0.001	***0.938
	47T	*6.9*	*1*	< 0.001	***0.979
	47F	*5.6*	*1*	< 0.001	***0.938
	49T	*8.1*	*1*	< 0.001	***0.99
	49F	*6.9*	*1*	< 0.001	***0.979
getClassName	2T	**1.5**	**144.3**	< 0.001	***0.183
	2F	*12.5*	*4.8*	< 0.001	**0.704
isHostNameLabel	3F	**4.1**	**6**	< 0.001	***0.007
compareLabels	6F	**3.2**	**6.2**	< 0.001	***0.034
	11T	*168*	*48.8*	< 0.001	***1
	11F	**1**	**14.8**	< 0.001	*0.37
isValidProtocol	4T	**4**	**6**	< 0.001	***0.015
	9F	*131.5*	*131.1*	< 0.001	***0.189
	11T	*122.9*	*40.8*	< 0.001	***0.863
getURLPrefix	4T	*123.7*	*25.9*	< 0.001	***0.935
	4F	**1**	**124**	< 0.001	***0.094
	7T	**12.3**	**124**	< 0.001	***0.221
validateEmail	7F	**9.2**	**11**	< 0.001	***0.813
	10T	**1**	**26.5**	< 0.001	***0.154
	12T	**1**	**26.5**	< 0.001	***0.154
	12F	**1**	**26.5**	< 0.001	***0.154
	14T	**1**	**64.6**	< 0.001	***0.04
	14F	**137.1**	**180**	0.02	*0.36
	20T	**1**	**44.3**	< 0.001	***0.15
	20F	*496.8*	*64.4*	< 0.001	***0.835
	24T	**1**	**53.6**	< 0.001	***0.156
	24F	**1**	**94.1**	< 0.001	***0.039
	27T	**11**	**274.1**	< 0.001	***0.019
	27F	**9.5**	**98.3**	< 0.001	***0.048
	31T	**1**	**94.1**	< 0.001	***0.039
	31F	**15.9**	**168.3**	< 0.001	***0.027
	34T	**1**	**94.4**	< 0.001	***0.039
	34F	*762.6*	*393*	0.008	***0.981
	36T	**15.9**	**168.3**	< 0.001	***0.027
	36F	**3**	**94.6**	< 0.001	***0.076
	41T	**9.4**	**98.3**	< 0.001	***0.048
	41F	**41.8**	**274.1**	< 0.001	***0.069
	43T	**1.3**	**167**	< 0.001	***0

Table 3.8: The results of the Wilcoxon rank-sum on numbers of fitness evaluations performed using student-supplied seeded and unseeded approaches. The mean number of fitness evaluations for successful trials using each approach are presented in columns **Seeded** and **Unseeded** respectively. The last column presents the sample's effect size using $\hat{A}_{12}$ statistic [117].

(b) Student-supplied Seeded Test Data

Method	**Branch**	**FitnessEvals**		***p*-value**	$\hat{A}_{12}$
		Seeded	**Unseeded**		
factorial	2T	**3.6**	**7.4**	0.004	***0.248
	2F	*1.7*	*1*	0.004	*0.583
gcd	8T	**7**	**15**	< 0.001	***0.1
	8F	**1**	**31.8**	< 0.001	***0.25
	16T	*1.2*	*1*	0.006	*0.575
	19T	*1.2*	*1*	0.006	*0.575
	21T	*1.2*	*1*	0.006	*0.575
binomialCoefficient	2T	**3.6**	**12.1**	0.005	***0.24
	4T	*279.5*	*158.3*	0.039	**0.689
	8T	**2.2**	**380**	0.039	***0
compareTo	1T	**1.5**	**43.6**	< 0.001	***0.143
	1F	*7.3*	*6*	0.002	***0.744
	3T	**16.6**	**76.8**	0.006	***0.257
	3F	**21.4**	**48.1**	0.007	***0.722
days_between	2T	**4.4**	**15**	< 0.001	***0
	4F	**1**	**11.5**	< 0.001	***0.25
	6T	*3.7*	*1*	< 0.001	***0.844
	6F	**2.1**	**15**	< 0.001	***0
	10F	**1**	**9.5**	< 0.001	***0.25
	12F	**1**	**12.5**	< 0.001	***0.25
	14F	**1**	**8.9**	< 0.001	***0.25
	16F	**1**	**12**	< 0.001	***0.25
	18F	**1.3**	**40**	0.004	**0.298
	28T	**3.7**	**120.6**	< 0.001	***0
	28F	*2.8*	*1*	< 0.001	***0.75
	32T	**48.4**	**206.3**	< 0.001	***0.103
	32F	*6.2*	*1*	< 0.001	***0.103
	34T	**49.2**	**206.3**	< 0.001	***0.103
	34F	**48.4**	**206.3**	< 0.001	***0.103
	39F	*6.2*	*1*	< 0.001	***0.416
	43T	**6.2**	**31.9**	< 0.001	***0.78
	43F	*6.2*	*1*	< 0.001	***0.78
	47T	*7.2*	*1*	< 0.001	***0.78
	47F	*6.2*	*1*	< 0.001	***0.78
	49T	*8*	*1*	< 0.001	***0.78
	49F	*7.2*	*1*	< 0.001	***0.78

Continuous of Table 3.8 - Student-supplied Seeded Test Data

Method	**Branch**	**FitnessEvals**		***p*-value**	$\hat{A}_{12}$
		Seeded	**Unseeded**		
getClassName	2T	**1**	**144.3**	< 0.001	***0.143
	2F	*22.2*	*4.8*	< 0.001	***0.761
protectQuotations	3T	**20.6**	**41.7**	0.006	***0.266
isHostNameLabel	3F	**4.3**	**6**	< 0.001	***0.02
compareLabels	6F	**2.7**	**6.2**	< 0.001	***0.051
	11T	**8.2**	**48.8**	< 0.001	***0.195
	13T	*1.4*	*1.1*	< 0.001	**0.681
	13F	**7.3**	**81.2**	< 0.001	***0.065
isValidProtocol	4T	**4**	**6**	< 0.001	***0.048
	9F	**32.8**	**131.1**	< 0.001	***0.069
	11T	**28**	**40.8**	< 0.001	***0.846
getURLPrefix	4T	*309.6*	*25.9*	< 0.001	***0.949
	4F	**1**	**124**	< 0.001	***0.094
	7T	**12.2**	**124**	0.01	***0.227
validateEmail	7F	**9.6**	**11**	< 0.001	***0.778
	10T	*43.3*	*26.5*	< 0.001	***0.227
	10F	**179.8**	**202.4**	0.008	**0.293
	12T	*43.4*	*26.5*	< 0.001	***0.234
	12F	*43.4*	*26.5*	< 0.001	***0.234
	14T	**45.1**	**64.6**	< 0.001	***0.118
	14F	**142.1**	**180**	0.002	***0.236
	20T	*48.8*	*44.3*	< 0.001	***0.229
	20F	*355.4*	*64.4*	< 0.001	***0.921
	24T	**50.5**	**53.6**	< 0.001	***0.233
	24F	**50.5**	**94.1**	< 0.001	***0.122
	27T	**55.2**	**274.1**	< 0.001	***0.074
	27F	**55.2**	**98.3**	< 0.001	***0.118
	31T	**50.5**	**94.1**	< 0.001	***0.122
	31F	**93.6**	**168.3**	< 0.001	***0.119
	34T	**51.1**	**94.4**	< 0.001	***0.132
	36T	**93.6**	**168.3**	< 0.001	***0.119
	36F	**55.7**	**94.6**	< 0.001	***0.169
	41T	**55.2**	**98.3**	< 0.001	***0.118
	41F	**76.9**	**274.1**	< 0.001	***0.104
	43T	**1**	**167**	< 0.001	***0

the use of the seeded search-based approach can significantly reduce the number of fitness evaluations required to cover complex branches. In any cases, the relevant success rate improves when using the seeded search-based approach. This can be due to the application of the human-supplied seeds which incorporate subtle knowledge about the program into the search process, and assist the search to effectively locate the test data of interest.

RQ4. Test Data Fault Finding Capability

The mutation system for Java programs, MuClipse [7] was used to assess the fault-finding capability of test data generated using each approach. MuClipse is a plugin for Eclipse [5], which generates different types of mutants for both traditional and class-level mutation testing automatically.

The fault-finding capabilities of generated test data were assessed based on their mutation scores. The Fisher exact test was performed to determine the statistical significance with confidence level set to 95%. Table 3.9 presents these results with significant p-values displayed in the last column. Table 3.10 shows the number of mutants generated for each program.

As evident from the Table 3.9, test inputs generated using the seeded approach have slightly lower mutation scores for programs *days_between*, *protectQuotations*, and *compareLabels*. This difference however, is not statistically significant as none of the corresponding p-values are within the significance threshold (< 0.05). Mutation score for *composeName* is undefined as MuClipse failed to generate any mutants due to the absence of any syntactic features (e.g. arithmetic operators, unary logic, etc) in the method that can be modified using the traditional operators.

In response to this research question, the analysis of the results suggests the use of the seeded search-based approach has no negative effects on mutation score. In fact, it can significantly increase the mutation score in certain cases. This could be due to the branch-covering characteristics of the seeded test data which adds more diversity to the generated test suites.

Table 3.9: Fault-finding capability of test suites generated using the seeded and unseeded search-based approaches, with p-values less than 0.05 diplayed in bold. Mutation Score is the percentage of mutants a set of test cases can detect. In all significant cases, the seeded test suites obtain higher mutation score compared to the unseeded test suites. (a) corresponds the crowd-sourced seededed test data, while (b) represents the student-supplied seeded test data.

(a) Crowd-Sourced Seeded Test Data

Project	**Method**	**Mutation Score**		***p*-value**
		Seeded	**Unseeded**	
Apache Commons	factorial	82.0	68.0	0.6
	gcd	84.0	83.0	0.9
	binomialCoefficient	79.0	68.0	0.3
	compareTo	**69.0**	**30.0**	**0.005**
Calendar	days_between	91.0	92.0	0.9
Chemeval	**isValid**	**89.0**	**3.0**	**< 0.001**
Daikon	getClassName	80.0	80.0	1.0
	protectQuotations	83.0	83.0	1.0
OpenJDK	**isHostNameLabel**	**85.0**	**22.0**	**< 0.001**
	compareLabels	94.0	95.0	1.0
	composeName	-	-	-
	isValidProtocol	97.0	78.0	0.5
	getURLPrefix	**82.0**	**27.0**	**0.008**
PuzzleBazar	**validateEmail**	**80.0**	**56.0**	**< 0.001**

(b) Student-Supplied Seeded Test Data

Project	**Method**	**Mutation Score**		***p*-value**
		Seeded	**Unseeded**	
Apache Commons	factorial	89.0	68.0	0.4
	gcd	85.0	83.0	0.8
	binomialCoefficient	78.0	68.0	0.4
	compareTo	**58.0**	**30.0**	**0.031**
Calendar	days_between	91.0	92.0	0.9
Chemeval	**isValid**	**88.0**	**3.0**	**< 0.001**
Daikon	getClassName	80.0	80.0	1.0
	protectQuotations	77.0	83.0	0.9
OpenJDK	**isHostNameLabel**	**73.0**	**22.0**	**< 0.001**
	compareLabels	67.0	95.0	0.1
	composeName	-	-	-
	isValidProtocol	95.0	78.0	0.5
	getURLPrefix	**82.0**	**27.0**	**0.008**
PuzzleBazar	**validateEmail**	**78.0**	**56.0**	**0.001**

Table 3.10: The total number of mutants generated for each method using MuClipse

Project	**Method**	**Mutants**
Apache Commons	factorial	57
	gcd	270
	binomialCoefficient	223
	compareTo	78
Calendar	days_between	574
Chemeval	isValid	129
Daikon	getClassName	15
	protectQuotations	36
OpenJDK	isHostNameLabel	110
	compareLabels	121
	composeName	0
	isValidProtocol	70
	getURLPrefix	40
PuzzleBazar	validateEmail	466

3.5 Threats to Validity

An important part of any empirical study is to consider the threats to the validity of the experiment. As this study is concerned with comparison of two different approaches to test data generation, it is essential to explore both internal and external validity of the experiment to ensure that the comparison is as fair as possible. This section discusses these potential threats and how these were addressed.

Internal validity emphasises on identifying the potential source of bias in the experimental design that could have affected the obtained results. In this experiment, one source of bias could originate from the stochastic behaviour of the meta-heuristic search algorithms employed in test data generation. The most reliable (and widely used) scheme for overcoming this threat is to perform the test data generation process using a sufficiently large data sample. In this study, experiments were repeated 50 times using the unseeded approach, and a variable number of times using the seeded approach depending on the supplied number of seeds for each program.

This was extensively explained in Section 3.3.4.

Another potential threat, which is a common drawback of human studies is the "learning effects". Participants tend to perform significantly better at the end of the study due to becoming more familiar with the task. This can bias the results obtained from the beginning of the study. Converse to the "learning effects" are the "fatigue effects", where human subjects tend to become tired towards the end of long studies or analysis, which could bias the obtained results. To mitigate these risks skewing the results, only correct responses from participants (both students and crowd-workers) were used to initiate the search process.

In practice, the values obtained for seeding may not be pre-filtered. Thus, filtering the answers obtained from the participants may cause an additional threat to validity. To investigate this threat, additional work is required to assess the quality of the seeded data obtained from various resources, and explore how these may affect the seeding approach in general. More details about this is presented in Chapter 6.

External validity is concerned with the extent to which the results of the experiment can be generalised or applied to real world data. A possible source of bias in this sense is the selection of the programs used in the empirical study. Due to the rich and diverse nature of programs, it is impractical to sample a sufficiently large set that represents all the characteristics of all possible programs. In this experiment, wherever possible, a variety of programming styles and sources were used. The empirical study drew on 14 Java methods comprising of 130 branches, providing a large pool of results from which to make observations. The selected Java methods included various primitive input types(string, integer and double) and were selected from 6 open source projects developed by real programmers.

A potential threat to external validity is due to the type of search-based algorithm used. The results obtained from analysing one particular search-based approach may not be applicable using other approaches. In this study, test data was generated using the AVM, and any patterns observed in these results may only be applicable to this particular method, and not to other search-based test data generation approaches in general.

Another source of bias is the use of CrowdFlower the crowd-sourcing

website. One concern about crowd-sourcing studies is the quality of the collected data due to the diversity of anonymous users, and their unknown level of experience. This may provide the opportunity for people to misreport the level of expertise or "game" the system by providing arbitrary judgments just s was fully described in Section 3.3.3.

Another threat to validity is the use of students as professionals, and generalising the outcome to the broad population. University students are often not appropriate representative of the general population with regards to a host of issues. Computer Science (CS) students are however an exception as they tend to be closer to the world of software professionals more closely than other students (e.g. psychology) are to the general population [113]. In particular, CS graduate students are so close to professional status that the differences are marginal. In fact, CS graduate students are technically more up to date than the "average" software developer who may not even have a degree in CS. For more information about this, the reader is referred to [113]. Another argument to this is that professionals with years of experience may solve a given problem better than appropriately prepared (graduate) students. Studies have however found that level of professional experience has little to do with competence [113].

The potential threats concerning the data's distribution type and the data sample size were mitigated using the non-parametric statistical measures. Assumptions regarding the normality of the samples can introduce a further sources of error into the study. The Fisher's exact test and the Wilcoxon rank-sum test were used to indicate the statistical significance. These are non-parametric statistical hypothesis tests that do not require any assumptions about the shape of the distribution.

Construct validity refers to the degree to which an experiment can subjectively measure a construct. This relates to the suitability of the employed measures in defining the performance of a technique. In this study, the performance of both seeded and unseeded approaches was empirically assessed in terms of three different criteria; branch coverage, efficiency and fault-finding capability. These metrics were merely used as a comparison measure for contrasting the seeded and unseeded approaches.

3.6 Conclusions

This chapter investigated a seeded search-based test data generation approach, in which the search process was incorporated with human knowledge that was provided in the form of sample test cases. An empirical study was conducted to collate examples of test inputs for a number of Java methods from human subjects. The subjects came from two broad groups: 29 students with beginner to advanced programming skills from the Department of Computer Science at the University of Sheffield, and 117 Internet users, with self-reported experience in computer programming, participating via CrowdFlower the crowd-sourcing website. The human-supplied values were used to seed the search-based test data generation process, and guide the search mechanism towards similar values.

The results of the empirical analysis revealed that the use of a seeded search-based approach can indeed improve the branch coverage as well as mutation score in a number of cases. In cases where branch coverage remained unchanged, the seeded approach performed with relatively higher efficiency.

Chapter 4

An Investigation into a Seeded Search-based Approach For Oracle Cost

4.1 Introduction

As discussed in the literature review Section 2.5.7, one of the sources of human oracle cost is the difficulty of reading machine-generated test inputs. Test data generated by automatic test input generators are often arbitrarily looking, and difficult-to-read values that are dissimilar to test inputs a human tester would normally generate. Manual evaluation of such values and interpreting the scenarios these arbitrary-looking values represent is a difficult and time consuming task.

It was shown in the previous chapter that seeding a search-based test data generation approach with appropriate human-supplied values would result in production of more branch-covering and fault-revealing test inputs. This chapter inspects whether the application of such an approach will have any influence on test data readability, and consequently any impact on test data oracle costs. This is investigated by an empirical study, in which test data readability is assessed in terms of manual test data evaluation time.

Programmers were invited to evaluate test cases generated using both seeded and unseeded approaches, while being timed during the process. The

study found that test inputs generated using the seeded search-based approach took significantly less time to evaluate for the majority of case studies. The accuracy of test input evaluation was also found to be significantly improved in a few cases. The key contribution of this chapter is therefore as follows:

> *The results of a human study in which test data generated using both seeded and unseeded search-based approaches were evaluated by human subjects. The results revealed cases in which the manual evaluation task was less time consuming and more accurate for test data generated using the seeded approach.*

This chapter is organised as follows. Section 4.2 describes the empirical study, depicting the experimental setup. Section 4.3 investigates the results, while Section 4.4 addresses the threats to validity. Finally, the conclusions of the chapter is presented in Section 4.5.

4.2 Experimental Study Methodology

An empirical study was performed in which test data produced in the previous experiment (Section 3.3.4) was used as the basis for a human evaluation task. The study was designed to record the time human testers required to determine the expected output of the 14 Java method against a set of test inputs. The main objective was to assess the time and accuracy human testers would require to manually evaluate the automatically generated test inputs by hand. The study consisted of the following major steps:

1. The selection of test inputs as the basis for the test data evaluation task.

2. The human study protocol regarding the information presented to the human subjects, and the responses obtained.

3. The selection of the participants.

The description of each stage is presented in the following sections in sequential order.

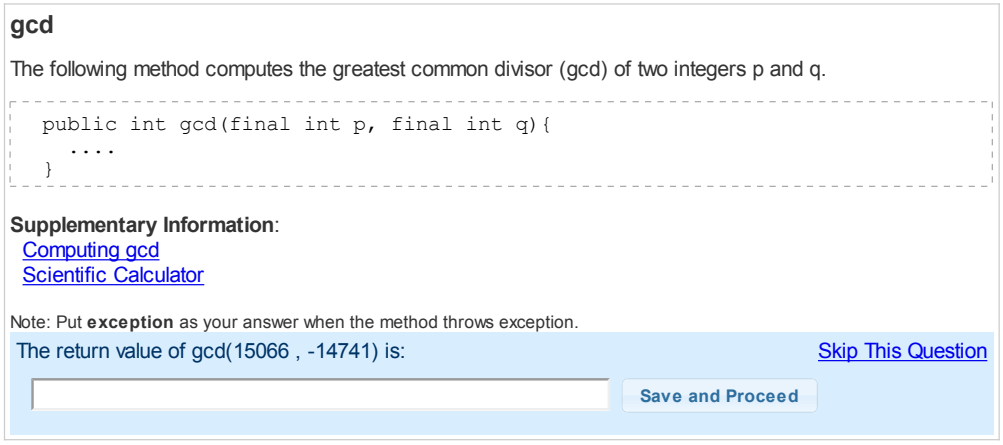


Figure 4.1: TCEvaluator - Presents a total of 8 questions (inolving a test cases) successively to each the participant

4.2.1 Test Input Selection

In the previous experiment, a large number of test suites were generated for the 14 Java programs listed in Table 3.1. These contained various numbers of test cases for each method depending on the number of branches, and the search's number of runs. To create a uniform experimental setup in the current study, it was necessary to select a fixed number of test inputs for each program for the evaluation phase. This was implemented using a randomisation function which would primarily extract all the different test cases of a test suite, and subsequently select a fixed number of test inputs among them. This function was set to select a total of 30 diverse test inputs from each of the seeded and unseeded sets. This allocated each method with a pool of 60 test inputs, to be evaluated by human subjects at the end of the evaluation task.

4.2.2 Human Study Protocol

This phase of the empirical study was concerned with assessing the time human subjects would require to manually evaluate automatically generated test inputs. This process was automated as a web application, referred to as *TCEvaluator*. The application firstly presented a brief description about the study, and requested the participant to specify their level of education

and their field of expertise. It then displayed the description of one of the 14 methods, and displayed 8 questions to participants to answer.

Each question requested the participant to supply an output for a randomly selected input according to the method's description. The output could be a boolean value (i.e. 'true' or 'false'), a string or an integer value depending on the method in the question. The participant was expected to enter the output in a provided text field. The time duration taken from the presentation of each question to the participant entering and saving their responses was recorded, with the response logged internally as "correct" if it matched the actual return value of the method.

The selection of each program was determined based on the number of evaluations the program had previously received. This was computed using a counter field, which was set to zero by default, and would increment every time the corresponding program was evaluated by the tester. Once the application launched, a query would fetch all the programs with the least number of evaluations from an underlying database used to store all the results. A randomisation scheme would then select a program from the fetched list.

As described previously in Section 4.2.1, each of the 14 methods were allocated with 60 different test inputs selected randomly from a larger set. For each question, a test input was selected at random from the pool of 60 inputs to be evaluated by the human tester. The main reason for randomisation was to mitigate bias that could be introduced by a fixed ordering of questions due to the possible "learning" or "training" effects.

Participants were presented with an equal number of seeded and unseeded test inputs (i.e. 4 from each category) to evaluate. The questions were displayed to the participant in a random order, and only one attempt was permitted to complete the study. In order to familiarise the participant with the case study, the first two questions were assigned as practice questions. The answers to these questions were not used in the data analysis presented in Section4.3.

Figure 4.1 shows a screenshot of the TCEvaluator with an example question. Once the "Save and Proceed" button was clicked, no further editing of answers was allowed.

4.2.3 Participant Selection

Human subjects for this experiment originated from two different groups: students from the Department of Computer Science, and crowd-workers recruited from the crowd-sourcing channel. All subjects were required to have some level of self-reported experience in computer programming. Students were approached via email invitations, while crowd-workers were contacted through the CrowdFlower website.

As discussed in the previous chapter, crowd-sourcing platforms are open to participants mis-reporting expertise levels or performing tasks randomly in order to earn money quickly. To avoid these biasing the final results, only a limited subset of the total participants were considered in the analysis of data. This selection was completed based on the correctness of each participant in answering the questions they were presented with. Since the study was not intended to challenge the participants level of ability, a programmer of even a basic level of competence should have been able to answer the majority of the questions correctly. On this basis, participants who failed to answer at least 50% of the questions correctly were discarded from the study. Thus only participants who evaluated at least 4 out of the 8 test cases "correctly" were considered as eligible, and their responses were considered in the data analysis presented in Section 4.3.

This selection scheme, implemented within the TCEvaluator, was used to provide the participant with a confirmation code if at least 50% of their responses were correct. Participants were then requested to paste the confirmation code into the text box displayed in CrowdFlower interface to claim their payment. Participants with less than 50% correct responses were not provided with the right confirmation code, and thus failed to receive any payments from CrowdFlower. This scheme was also applied on students in order to identify and discard ineligible participants from the study.

The labour fee for evaluating 8 test cases for each method (consisting of 8 questions) was specified as 25 cents for crowd-workers. This amount would be given to the participant upon completion of the evaluation task. The total cost for this experiment was 512.05 USD including the additional markup for the CrowdFlower labour costs. There was no labour fees specified for

Table 4.1: The number of total and usable responses obtained from students and crowd-workers for each case study. The second column displays the number of case studies (Java methods) evaluated by each group of participants. Only 5 Java methods were evaluated by students in total. Each method was evaluated by a total 4 eligible students. This allocated each method with a total of 32 (i.e. 4×8) responses (including the responces to the first two training questions). The sixth column displays the number of remaining responses for each method (excluding the first two). A total of 84 eligible crowd-workers evaluated each method, which resulted in 672 responses.

Groups	**Case Studies**	**Participants**		**Responses**		**Fees**
		All	**elig**	**All**	**Excl.1-2**	
Students	5	4	4	32	24	20×10 GBP
Crowd	14	100	84	672	500	512.05 USD

students. However, 10 students (out of the total 20), chosen at random, were awarded with a £20 voucher token.

4.2.4 Usable Judgements

A common source of bias in human studies is the potential "learning effects", where participants tend to perform significantly better at the end of the study due to becoming more familiar with the task. To mitigate these learning effects biasing the results, the answer to the first two questions (which were assigned as practice questions) were not used in the analysis of results. These two responses could have been for seeded or unseeded test inputs. At the end, 250 usable responses were collated involving inputs generated by each approach for each method (500 for each method in total) for data analysis presented in Section 4.3.

4.2.5 Basic Definitions

Accuracy Score. This is defined as the percentage of test inputs for which participants entered the correct outputs for each method in the human evaluation task.

Cognition Time. This refers to the time participants required to determine the expected output of a method for a given input.

4.2.6 Research Questions

The research questions to be answered by the empirical study are as follows:

RQ 1. Test Data Accuracy Score. This research question establishes whether the use of the seeded search-based approach can significantly increase the *accuracy score* of generated test inputs. To inspect this, the percentage of participant responses in which the correct output was entered for inputs generated using each approach will be computed and compared.

RQ 2. Test Data Cognition Time. This research question inspects whether the use of the seeded search-based approach can significantly reduce test inputs *cognition time*. To answer this, the time participants required to provide outputs for inputs generated using each approach will be computed and compared. This is performed on two selections of data: all the collated responses, and only the responses in which participant provided the correct output (i.e. correct judgements).

4.3 Experimental Results

This section analyses these results, evaluating each research question.

RQ1 - Test Data Accuracy Score

The main objective of this research question was to determine and compare the accuracy of humans in manually evaluating seeded and unseeded test inputs. To inspect this, the accuracy score of test inputs generated using each approach for each method was computed. As there were two different groups of participants (students and crowd-workers), Table 4.2 displays these results separately for each group. The data obtained from students was only limited to a few case studies due to the limited number of student volunteers for this task.

Table 4.2: The results of the Fisher's exact test on the percentage of correct judgements (accuracy score) obtained from (a) crowd-workers, and (b) students. Judgments obtained from crowd-workers correspond to the crowd-sourced seeded test data and unseeded test data generated previously. Similarly, judgments obtained from students corespond to the student-supplied seeded and unseeded test data generated in the previous experiment. These are presented under column names **Seeded** and **Unseeded** in both tables (a) and (b). Data obtained from students are only limited to a few case studies due to the low number of participants. A p-value in bold face indicates the cases where the use of the seeded approach had positive effects on accuracy score. Italic face indicates cases where the use of the seeded approach was significantly detrimental (i.e. *isValid*).

(a) Crowd-Sourced Judgements

Project	**Method**	**Accuracy Score**		***p*-value**
		Seeded	**Unseeded**	
Apache Common	factorial	89.6	94.4	0.699
	gcd	87.6	70.0	0.106
	binomialCoefficient	78.4	65.6	0.213
	compareTo	85.2	79.2	0.595
Calendar	**days_between**	**92.3**	**17.9**	**< 0.001**
Chemeval	**isValid**	*68.8*	*94.8*	**0.019**
Daikon	getClassName	94.4	83.6	0.362
	protectQuotations	89.2	84.4	0.694
OpenJDK	isHostNameLabel	93.2	84.0	0.434
	compareLabels	**77.2**	**32.2**	**< 0.001**
	composeName	95.2	76.0	0.099
	isValidProtocol	94.0	83.2	0.361
	getURLPrefix	78.8	78.0	0.946
PuzzleBazar	validateEmail	78.8	95.2	0.168

(b) Student Judgements

Project	**Method**	**Accuracy Score**		***p*-value**
		Seeded	**Unseeded**	
Daikon	getClassName	100.0	91.7	1.000
	protectQuotations	100.0	100.0	1.000
OpenJDK	isHostNameLabel	80.0	93.3	0.795
	getURLPrefix	87.5	80.0	1.000
PuzzleBazar	validateEmail	92.9	100.0	1.000

The Fisher's exact test with confidence level of 95% was performed to verify the statistical significance. This is presented in the last column of Table 4.2 with significant p-values displayed in bold. As evident from these results, the Fisher exact test indicated no significant differences between the seeded and unseeded approaches for the majority of case studies. For one of the numerical computation methods (i.e. *days_between*) and one of the string conversion routine (i.e. *compareLabels*) the seeded search-based test data revealed a significant improvement in accuracy score. For one of the string validation methods (i.e. *isValid*), however, the accuracy score of the seeded test data revealed to be detrimental.

As discussed in Section 3.3.1, the method *isValid* ensures the validity of a string input as a CAS number. A valid CAS number is defined as a sequence of at least 5 digits separated by hyphens into three distinct parts, with the last digit serving as a check sum. Manual verification of this method against the generated set of test inputs can be a complex task since the tester must ensure each part of the string conforms to the specified format. This involves checking that the last digit correlates correctly with remaining digits based on some manual calculations.

Test data generated using the unseeded approach for the *isValid* method only included "invalid" values for a CAS number. These were arbitrarily sequences of characters such as "*q5'fy#ap%FAUm*"that could be instantly recognised as "invalid". Test data generated using the seeded approach, however, did enclose a number of "valid" test inputs such as "*7732-18-5*". It could be for this reason that the accuracy score for the seeded search-based test inputs revealed to be detrimental.

There was no evidence to suggest that test inputs generated using the seeded search-based approach reduce accuracy for any of the numerical computation and string conversion routines. The accuracy score of the seeded search-based test data revealed to be detrimental for string validation routines such as *isValid*. This was due to insufficiency in the number of valid test cases generated (i.e. low branch coverage) for this method.

In answer to this research question, therefore, the evidence suggests that the application of the seeded search-based approach can improve the accuracy of test input evaluation for programs that perform extensive numerical

calculations (such as *days_between* and *compareLabels*).

RQ2 - Test Data Cognition Time

The key objective of this research question was to assess and compare the time human subjects required to manually evaluate seeded and unseeded test data. To inspect this, the cognition time of each test input was recorded in the human empirical study using a Javascript running in the webpage. The timer started as soon as the test input was displayed to the participant, and stopped whenever they entered an output for the displayed input. This allocated each judgement collated from the participant with a timing value.

The average of all timing values associated with seeded and unseeded test inputs were computed separately for each program. This computation was made on two different selections of the collated data, the first selection included all the responses regardless of their correctness, and the second selection enclosed only correct responses in which the participant had entered the correct output for the given input.

Table 4.3 shows mean times for all judgements made by participants, while Table 4.4 shows the mean times for correct judgements only (i.e where the participant entered the correct output for the input). Statistical significance was tested for using the Wilcoxon rank-sum test at a confidence level of 95%. For each of these selections, the average cognition time for the seeded test data revealed to be significantly lower compared to the unseeded test data for 8 and 9 of the 14 case studies respectively.

Figure 4.2 shows box and whisker plots of the times recorded for correct judgements. The plots show superiority of the seeded approach in producing test input that require lower cognition times for method *gcd*, *getClassName*, *composeName*, *days_between* and *compareTo*. This was confirmed by computing the effect size using Vargha and Delaney's $\hat{A}_{12}$ statistic [117], as recorded in Table 4.3. Methods *gcd*, *getClassName*, and *composeName* involved large effect sizes. A further 2 methods *protectQuotations* and *compareLabels* experienced medium effect sizes, while the effect size was small for the *days_between* method.

There was no significant difference between seeded and unseeded test

Table 4.3: Cognition time of seeded and unseeded test data according to **all** the judgements obtained from (a) Crowd-workers, and (b) Students

(a) Crowd-Sourced Judgements

Project	**Method**	**Cognition Time**		***p*-value**	$\hat{A}_{12}$
		Seeded	**Unseeded**		
Apache Common	**factorial**	*11.0*	*9.9*	**0.003**	* **0.576**
	gcd	**18.2**	**60.0**	< **0.001**	*** **0.280**
	binomialCoefficient	**24.5**	**30.6**	**0.049**	0.551
	compareTo	**22.9**	**34.5**	< **0.001**	* **0.393**
Calendar	days_between	51.3	102.5	0.303	0.529
Chemeval	**isValid**	*43.4*	*6.2*	< **0.001**	*** **0.846**
Daikon	**getClassName**	**9.8**	**15.4**	< **0.001**	*** **0.258**
	protectQuotations	**14.4**	**23.0**	< **0.001**	** **0.327**
OpenJDK	**isHostNameLabel**	**7.7**	**10.2**	**0.034**	0.445
	compareLabels	**28.2**	**50.4**	< **0.001**	* **0.364**
	composeName	**15.4**	**27.7**	< **0.001**	*** **0.276**
	isValidProtocol	9.1	8.5	0.628	0.513
	getURLPrefix	*26.3*	*18.8*	< **0.001**	* **0.632**
PuzzleBazar	**validateEmail**	*9.3*	*6.8*	< **0.001**	* **0.611**

(b) Students Judgements

Project	**Method**	**Cognition Time**		***p*-value**	$\hat{A}_{12}$
		Seeded	**Unseeded**		
Daikon	getClassName	8.9	10.2	0.378	* **0.389**
	protectQuotations	9.6	10.6	0.932	0.486
OpenJDK	isHostNameLabel	9.1	8.1	0.128	** **0.688**
	getURLPrefix	36.7	56.6	0.799	0.535
PuzzleBazar	validateEmail	8.8	9.6	0.630	* **0.562**

Table 4.4: Cognition time of seeded and unseeded test data based on the **correct** judgements obtained from (a) Crowd-workers and (b) Students

(a) Crowd-Sourced Judgements

Project	**Method**	**Cognition Time**		***p*-value**	$\hat{A}_{12}$
		Seeded	**Unseeded**		
Apache Common	**factorial**	*10.7*	*8.7*	**0.001**	* **0.588**
	gcd	**17.7**	**60.9**	< **0.001**	*** **0.204**
	binomialCoefficient	**23.5**	**29.0**	**0.023**	* **0.570**
	compareTo	**21.9**	**36.2**	< **0.001**	* **0.370**
Calendar	**days_between**	**42.0**	**54.9**	**0.032**	* **0.391**
Chemeval	**isValid**	*52.9*	*6.2*	< **0.001**	*** **0.903**
Daikon	**getClassName**	**9.2**	**14.4**	< **0.001**	*** **0.233**
	protectQuotations	**14.6**	**20.0**	< **0.001**	** **0.326**
OpenJDK	**isHostNameLabel**	**7.4**	**10.8**	**0.003**	* **0.417**
	compareLabels	**24.7**	**37.1**	< **0.001**	** **0.337**
	composeName	**14.9**	**24.7**	< **0.001**	*** **0.273**
	isValidProtocol	8.9	8.4	0.438	0.521
	getURLPrefix	*25.3*	*17.7*	< **0.001**	** **0.661**
PuzzleBazar	**validateEmail**	*8.9*	*6.4*	< **0.001**	* **0.612**

(b) Students Judgements

Project	**Method**	**Cognition Time**		***p*-value**	$\hat{A}_{12}$
		Seeded	**Unseeded**		
Daikon	getClassName	8.9	10.2	0.378	* **0.389**
	protectQuotations	9.6	10.6	0.932	0.486
OpenJDK	isHostNameLabel	9.1	8.1	0.128	** **0.688**
	getURLPrefix	36.7	56.6	0.799	0.535
PuzzleBazar	validateEmail	8.8	9.6	0.630	* **0.562**

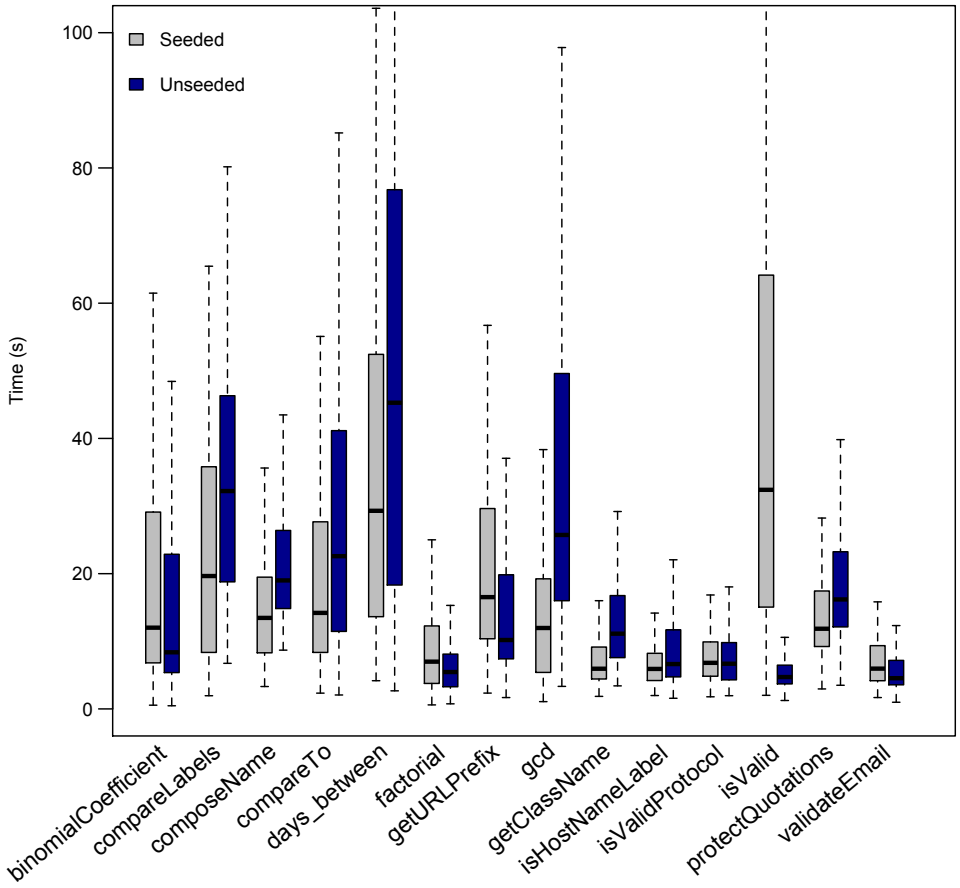


Figure 4.2: Box and whisker plots displaying the timing distribution for performing correct evaluations by crowd-workers on seeded and unseeded test inputs of each method. The centre line represents the median and the box the distribution of the data between the upper and lower quartiles. The upper and lower whiskers represent the minimum and maximum value (excluding outliers).

data for the *isValidProtocol* method. However, the cognition time of the seeded test data revealed to be significantly detrimental for methods *factorial*, *isValid*, *getURLPrefix* and *validateEmail*, with effect sizes of large for *isValid*, medium for *getURLPrefix* and small for the rest.

In answer to this research question, therefore, the evidence suggests that the use of the seeded search-based approach can reduce the cognition time of test inputs generated for numerical computation routines and string conversion routines. For string validation methods however assessing the effects of the seeded approach on cognition time is not always practical due to the low number of valid test cases generated for these programs. As described in the previous chapter, the majority of test cases generated for validation programs (such as *validateEmail* were invalid values covering invalid branches (i.e "*f#p%F@}UM%5.*6ZY*"). Manual evaluation of such values seem to be a straight forward task for humans.

4.4 Threats to Validity

As described in the previous chapter, a major source of bias in human empirical studies is the "learning effects". To alleviate these risks, the first two input evaluations performed by each participant were discarded from the analysis of results. To reduce the potential "fatigue effects", the number of test inputs the participant were requested to evaluate was kept as low as 8. The timing data obtained indicated that on average, each participant spent just under 3.5 minutes on a questionnaire, indicating that the study was not particularly complex or tedious and thus unlikely to be subject to fatigue effects.

The main step to mitigate both learning and fatigue effects was to randomise the questions. The test inputs that formed the basis of each question were selected at random from a pool of 60 test inputs, which were selected at random from the overall set of inputs generated using each approach. Moreover, the order in which the questions appeared to the participant was also randomised. These issues were fully discussed in Sections 4.2.1.

Another potential threat, is the quality of the data collected from CrowdFlower due to diversity of anonymous users, and their unknown level of

experience. As previously explained, crowd-sourcing studies are commonly open to people mis-reporting their level of expertise just to gain money. To mitigate this concern, the responses from participants who evaluated less than 4 out of 8 test inputs accurately (i.e. accuracy $< 50\%$) were discarded from consideration in the data analysis. This issue was extensively reviewed in Section 4.2.3.

Another potential threat to validity is the use of students as professionals, and generalising the outcome. As mentioned in previous chapter, CS students are known to closely represent the software professionals, and thus the potential discrepancy would be marginal.

Finally, to mitigate risks regarding the type of distribution and the normality of timing data, Fisher's exact test and the Wilcoxon rank-sum test were used to find the statistical significance. The data effect size was also checked using the Vargha and Delaney's $\hat{A}_{12}$ statistic. These are all non-parametric tests that do not rely on data belonging to any particular distribution, mitigating the introduction of further potential sources of error into the study.

With regards to construct validity in this part of the study, the performance of each approach was mainly assessed in terms of test data evaluation time. This measurement overlook the familiarity of human participants with the case studies, their experiences, or their IQ skills in comprehending the task. To mitigate these risks, the allocation of all the test cases to the participants for evaluation was purely based on random. In addition, the non-parametric statistical test were used to determine the significance in each case.

4.5 Conclusions

This chapter presented an empirical investigation into a seeded search-based test data generation approach, in which the search mechanism was adapted to commence with an appropriate human-supplied value. In the previous experiment, human subjects with self-reported competency in computer programming were invited to provide samples of test cases for a set of Java programs. These values were then seeded into the data generation process

to guide the search towards similar values. Test inputs generated using this approach were expected to have improved readability and thus reduced evaluation time.

This chapter investigated this hypothesis, through the use of another empirical study, in which human subjects were invited to evaluate a set of seeded and unseeded test inputs for a Java program. The subjects were selected from two different groups: students in the department of computer science in the University of Sheffield, and Internet users from the CrowdFlower website. The time each participant required to provide an output for the given input was recorded during this process.

The results of this study revealed that test data generated using the seeded search-based approach was less time-consuming to evaluate in several case studies. In addition, the accuracy of test input evaluation performed by human testers was also improved in a few cases. The study thus concluded that seeding a search-based test data generation approach with sufficient amount of human knowledge (in the form of samples test cases) can indeed enhance the readability of resultant test data, and thus reduce the qualitative human oracle costs.

Chapter 5

Test Data Generation Using A Language Model

5.1 Introduction

It was discussed in the previous chapters that seeding the search-based test data generation process with human-supplied values can improve the readability of resultant test cases for certain programs. The application of this seeding strategy is however subject to presence of a permanent resource (e.g human tester) that can manually supply appropriate inputs to seed the search mechanism. This chapter presents a novel approach that can automatically generate readable test inputs for string data types. This approach incorporates the search-based test data generation with a statistical natural language model that can assess and improve the readability of string inputs.

A language model assigns a probability score to a string estimating the chance of that string occurring in the language it models. This chapter shows how this probability score can be employed as an additional component for the fitness function, and guide the search-based test data generation process to produce more readable test inputs. Language models are most frequently used in natural language processing tasks [76] including machine translation [74], where they attempt to improve the fluency of machine translated texts such as predictive text in mobile phones. In speech processing [63] these are used to assist a speech recogniser evaluate how likely a word sequence

is, and thus make the right assumption when two different sentences sound the same. Language models have recently been applied to develop a code suggestion tool that makes use of the existing suggestion facility in the Eclipse IDE [55].

The capabilities of the language model approach in generating readable string inputs were investigated using a human empirical study. Human subjects with self-reported competency in computer programming were invited via the CrowdFlower website to evaluate tests cases for a series of 17 case studies from open source projects. Test cases were generated using both a conventional search-based approach, and the language model informed approach. The study aimed to assess the time participants required to manually evaluate test inputs of each approach. The results revealed that test inputs generated using the language model approach took significantly less time to evaluate for 10 case studies. The accuracy of participants in evaluating test inputs of the language model approach was also found to be significantly improved for 3 case studies.

The contributions of this chapter, therefore, are as follows:

1. Introduction of a technique for incorporating the automatic test input generation process with a statistical language model to generate readable branch-covering string inputs.

2. The results of a human study in which test data generated using the conventional search-based approach and the language model informed approach are evaluated by human subjects. The analysis reveals cases where human evaluation task is less time consuming and more accurate for test data generated using the language model approach

The chapter begins by describing the operation of the language model used in this study (Section 5.2), and how it is incorporated into the search-based test data generation process (Section 5.3). Section 5.4 then introduces the methodology used in the human study, while Section 5.5 presents the results. Section 5.6 describes the threats to validity, and finally Section 5.7 concludes the chapter.

5.2 Language Models

A statistical language model assigns a probability score to a string by estimating the likelihood of that string occurring in a natural language (e.g. English, Spanish or Japanese). An accurate language model would therefore calculate higher probability scores to strings that resemble well-formed words, such as "*software*"and lower scores to strings that do not, such as "*0NytRV8**".

The main applications of language models are in natural language and speech processing for a wide range of tasks, including machine translation [74], automatic speech recognition [63] and information retrieval [107]. Language models have also been adapted to simulate programming languages and used on software engineering tasks [56]. The majority of applications use word-based language models, which simulate the language as sequences of words. In this thesis, a character-based language model will be used, where the language is represented as a sequence of characters.

A character-based language model computes the probability of a string by analysing a collection of documents known as a *corpus*. The probability of the string str, of length n, is estimated by evaluating the number of instances the string occurs in the corpus (str_sum), divided by the total number of possible strings of the same length ($length\_n\_sum$) within the corpus, so $P(str) = str\_sum/length\_n\_sum$. This probability is estimated using the chain rule of probability as explained next.

Let the string str be a sequence of n characters $(c_1, c_2, c_3, ..., c_n)$. Using the chain rule, the probability of each character c_i is calculated based on the characters that precedes it in $(c_1, c_2, c_3, ..., c_n)$. This is computed using the following formula:

$$\begin{aligned} P(c_1, ..., c_n) &= P(c_1)P(c_2|c_1)P(c_3|c_1, c_2)...P(c_n|c1, c_2, ..., c_{n-1}) \\ &= \prod_{i=1}^{n} P(c_i|c_1, ..., c_{i-1}) \end{aligned} \tag{5.1}$$

where $P(c_i|c_1, ..., c_{i-1})$ is the probability of character c_i following the sequence $(c_1, ..., c_{i-1})$.

However, many of these sequences will not be found even in an extremely large corpus, making these probabilities impossible to estimate directly. If we consider the English language for example which has 26 characters in total (i.e. ignoring case, punctuation and whitespace), the total number of 6 character sequences is over 11 million.

To resolve this predicament, language models approximate the probability of strings by combining the probabilities of shorter sequences. These sequences have more reliable probabilities since they can generally be referenced directly from the corpus. One approach is to estimate the probability of each character based on the character that immediately precedes it:

$$P(c_1, ..., c_n) \approx \prod_{i=1}^{n} P(c_i|c_{i-1}) \tag{5.2}$$

This type of language model is known as a bigram model. However, even when using a bigram model some pairs of characters will not be seen in the corpus. In such situations, The probabilities of individual characters (i.e. $P(c_i)$) are estimated using smoothing and back-off techniques (for more details, the reader is referred to the references [63, 67]).

In general, longer strings are less likely to occur than shorter ones, and the language models assign them lower probabilities. To avoid bias in favour of shorter strings the probability generated by the language model is normalised by taking the geometric mean, i.e. the score assigned to a string, $score(c_1^n)$, is computed as:

$$score(c_1, ..., c_n) = P(c_1, ..., c_n)^{\frac{1}{n}} \tag{5.3}$$

Figure 5.1 shows the scores assigned by the bigram language model to the strings "*software*"and "*0NytRV8**".

It should be noted that the language model used in this experiment was implemented by Mark Stevenson. The SRILM toolkit [108] was used to learn the model and the text used to train it was an electronic version of the classic novel *Moby Dick* [85] downloaded from Project Gutenberg[1]. This text is freely available and contains 215, 133 words and 1, 235, 150 characters, which is more than adequate for training a language model.

[1] *http://www.gutenberg.org/ebooks/2701*

Bigram	Probability	Source
so	0.09015836	Direct
of	0.10778375	Direct
ft	0.17874846	Direct
tw	0.02277456	Direct
wa	0.18811646	Direct
ar	0.10431042	Direct
re	0.22745642	Direct

$P(\text{`software'}) = 0.14317363$

Bigram	Probability	Source
0N	0.00001776	Inferred
Ny	0.00001399	Inferred
yt	0.07976989	Direct
tR	0.00005748	Direct
RV	0.00771776	Direct
V8	0.00000056	Inferred
*8**	0.00000073	Inferred

$P(\text{`0NytRV8*'}) = 0.00046853$

Figure 5.1: Computing language model probabilities for two strings. The word "*software*" receives a higher probability than the random string "*0NytRV8**". For "*software*", all bigram probabilities can be found directly in the corpus, whereas some bigrams for "*0NytRV8**"are not present and are inferred from probabilities computed for each individual characters of the bigram separately.

5.3 Incorporating a Language Model Into Search-Based Test Input Generation

One of the main features of the search-based test data generation techniques is the formulation and application of a *fitness function* that provides guidance to the search algorithm. The common test goal in structural testing is branch coverage. As described in the literature review Section 2.4.1, the fitness function for this criterion consists of two major components *approach level* (AL) and *branch distance* (BD). The fitness function employed in the conventional search-based test data generation attempts to generate any inputs that can cover individual branches of the PUT. As a result, the generated test data can be arbitrarily looking values that are difficult to understand from a human perspective.

For instance, the input value "*#qp}ˆbkJ';_irg*" was generated for the *toCamel* method (in Figure 5.2) during the experiments reported later in Section 5.5. The method *toCamel* converts a string input to the camelCase format using the under_scoring style of joining words, where the first letter of each word (bar the first) is a capital letter. The conversion process involves finding each underscore in a string, removing it, and capitalising the following character. Based on the method's description, the correct output of this input is "*#qp}ˆbkJ';Irg*", which is not a instantly recognisable task. In practice, readable strings such as "*my_string*" would be preferred from a human perspective, and should be automatically generated.

The language model technique incorporates a statistical language model into the conventional fitness function designed for structural test data generation. The language model probability scores can be viewed as a measure of "likeness" or similarity of a string to natural words. As such, these scores can be used to form an ideal output of the fitness function in order to guide the search towards more natural and inherently readable string inputs. In this process the language model probability scores start to impact the search mechanism once an input that covers a branch is discovered, as shown in Figure 5.2.

Prior to locating a branch-covering input, the language model fitness component (LM) is always 1, and is added to AL and BD. Once a branch-

```
1  private  String toCamel(String str) {
2   StringBuffer sb = new StringBuffer();
3   boolean wasUnderline = false;
4   for (int i = 0; i < str.length(); i++) {
5    char c = str.charAt(i);
6    if (c == '_') {
7     wasUnderline = true;
8     continue;
9    }
10   if (wasUnderline) {
11    sb.append(Character.toUpperCase(c));
12    wasUnderline = false;
13    continue;
14   }
15   sb.append(Character.toLowerCase(c));
16  }
17  return sb.toString();
18 }
```

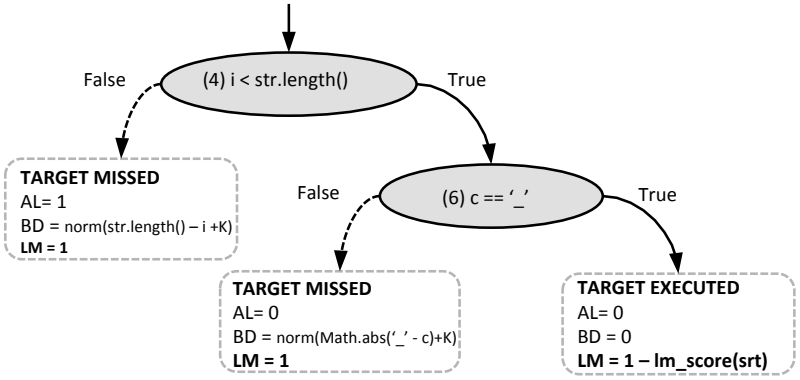


Figure 5.2: The process of fitness evaluation for generating a readable string that covers the true branch predicate c ==‘-’ in line 6 of the *toCamel* method. The fitness function consists of the conventional approach level (AL) and branch distance (BD) components (K is a positive constant set to 1 in this study), plus an additional metric (LM) obtained from the language model. LM is set to 1 while the search is attempting to cover the branch. Once a string input `str` covering the branch is discovered, the fitness function assigns probability scores (between 0 and 1) to the string using *lm_score*, generated for `str` by the language model. The higher the value of *lm_score*, the lower the value of LM, and the higher the similarity `str` has with strings in the language.

covering input is found – the point at which the search would normally terminate – the search process instead continues to optimise the input for the language model score. With the AL and BD scores both 0, the LM component returns $1 - lm(str)$, where str is the string input and $lm(str)$ is the language model function that returns a probability score for the string str. In other words, lower values of LM reflect "better" strings. The use of the LM component in the fitness computation almost prevents the search to reach the global optimum. The search must therefore be stopped at some suitable fitness evaluations limit. In this study, 100,000 fitness evaluations is used as the termination criterion.

In contrast to the value "*#qp}ˆbkJ';_irϑ*" generated for the *toCamel* method using the conventional approach, the language model approach is capable of generating strings such as "*inererof_yo*" in the experiments. The main objective of this chapter is to compare the two approaches in terms of the time required to manually evaluate the resulting test inputs. This will be investigated using a human empirical study detailed in the next section.

5.4 Experimental Study Methodology

An empirical study was performed to assess the time required for humans to manually evaluate string inputs generated using both the conventional approach and language model approach. The key objective was to investigate whether the manual evaluation task would be less time consuming and more accurate when the test data was generated with the assistance of a language model. This was exploited using a human empirical study consisting of following essential phases:

1. The selection of the case studies as the basis for test data generation.
2. The process of test data generation for string inputs using the language model approach and the conventional non-informed approach.
3. The selection of test inputs as the basis for the human evaluation task.
4. The human study protocol including the information presented to the human subjects, and the responses obtained.

Table 5.1: Case Studies

Project	Class	Methods	Branches
Bots'n'Scouts	de.botsnscouts.util.H	lesseqString	2
CodeHaggis	net.sf.haggis.actions.stringConverter.StringConverter	toCamel	6
Daikon	daikon.split.SplitterJavaSource	getClassName	2
		protectQuotations	4
Germoglio	P492	translate	2
Jake	org.jakedb.UserManager	isValidUsername	6
JavaMail	javax.mail.internet.InternetAddress	isSimpleAddress	2
		isGroup	4
JOX	com.wutka.jox.JOXBeanOutput	stripName	6
Muffin	sdsu.util.SimpleTokenizer	containsChar	4
OpenJDK	com.sun.jndi.dns.DnsName	isHostNameLabel	4
	com.sun.jndi.toolkit.url.GenericURLContext	composeName	4
PuzzleBazar	com.puzzlebazar.client.util.Validation	validateEmail	24
Rife	com.uwyn.rife.tools	capitalize	4
		encodeClassname	2
		needsUrlEncoding	6
Subsonic	net.sourceforge.subsonic.service.SearchService	containsIgnoreCase	4

5. The selection of the participants.

5.4.1 Case Studies

A total of 17 Java methods with string arguments were selected from 12 open source projects. A summary is presented in Table 5.1. One of the key points to consider in making this selection was that each case study would be the subject of human evaluation, and thus, it is important that the operation of each method is simply understood from no more than a few lines of text. The main reason for avoiding complicated methods was to mitigate the "fatigue effects" described in the previous chapter. To further mitigate fatigue effects biasing the results, the case studies were selected from simple Java methods with few branches. This was also reflective of good Java programming style, where short methods are best practice [2]. The case studies summarised in Table 5.1 are discussed in detail below:

Bots'n'Scouts is a multiplayer game on Robot Racing. The method *lesseqString*, selected from this project checks whether a string argument a is lexicographically less than or equal to another string argument b. If so, it returns true, otherwise it returns false.

CodeHaggis is an Eclipse code generation plugin, from which the method *toCamel* was selected. This method converts a string to a camel case format,

by firstly converting the whole string to lower case, then changing every character preceded by an underscore to its uppercase version, and finally removing all occurring underscores from the string.

Daikon is an invariant generator and detector tool, used for reporting likely program invariants. Two methods were selected from its source code: The *getClassName* method takes a string input and deduces a Java class name from it based on the final occurrence of the dot character in the string. The method *protectQuotations* takes a string input and places a backslash in front of each quotation mark.

Germoglio is a compilation of solutions to various programming problem contests. The method *translate* was selected from this project. This method translates a word string to 'Pig Latin' – a game of alterations in English language, in which words that start with a vowel (A, E, I, O, U) will be altered to have 'ay' appended to the end of the word, and words that start with a consonant will be changed to have their first character moved to the end of the word, followed by an 'ay'. For example, the string 'test' will be converted to 'esttay' and 'evaluation' will be changed 'evaluationay'.

Jake is a project that connects online resources such as academic journals and scholars using its large database. The method *isValidUsername*, selected from this project, checks whether the string argument is a valid user name, by ensuring it consists of at least three characters, and that each character is either alphabetic or numeric.

JavaMail is a Java API that provides various frameworks for emailing and messaging applications. Two methods were selected from this project: The method *isSimpleAddress* checks whether the string argument is a valid URL address. A valid URL is considered as a string that does not contain any forbidden characters including opening and closing brackets (round and square), colon, semicolon, less-than and greater-than symbols, backslash and comma. The method *isGroup* checks whether a string is a group address according to RFC822 standards, by ensuring it contains a colon and ends with a semi-colon.

JOX is project that contains a series of Java libraries, which facilitate transferring data between XML documents and Java beans. The method *stripName*, selected from this project, returns the lowercase version of its

string argument, with dash, underscore, dot, and colon characters removed.

Muffin is an internet filtering system which supports removing cookies, terminating GIF animations and eliminating advertisements. The method *containsChar*, selected from this system, tests if a character argument exists in another supplied string argument.

OpenJDK is the well-known open-source implementation of the Java programming language, consisting of the Java Class Library and the Java compiler. Two methods were selected from this project: The method *isHostNameLabel* takes a string as an argument, and returns true if the string is a valid host name. A valid hostname is considered as a string of which the first and the last characters are alphanumeric. The remaining characters may be alphanumeric or hyphens. The method *composeName* takes two strings $name$ and $prefix$ as arguments. If one of $name$ or $prefix$ are null or empty, the method returns the null or empty argument, otherwise it returns $prefix$ appended by a forward slash followed by $name$.

PuzzleBazar is a web-based system for creating, uploading and playing various puzzles, including learning tools and tutorials. The method *validateEmail* selected from this platform checks whether the string argument is a valid email address.

Rife is a content management framework used for developing web application in Java. Three methods were selected from this framework: *capitalize* takes a string and converts the first character to upper case if it is a lower case letter. The method *encodeClassname* takes a string and converts it to a valid Java class name, by replacing any characters that are not letters, digits or underscores with underscores. The method *needsUrlEncoding* inspects whether or not its string argument requires encoding by checking its validity as a URL string. In this case, a valid URL is considered as a string of which, each consisting character is either an alphabetical letter (upper or lower case), a digit or any of these characters: dash, underscore, dot, and asterisk. Any other character outside this category is not permitted.

Finally, *Subsonic* is a web-based music and video streaming application which allows sharing and listening to music online. The *containsIgnoreCase* method selected from this application, checks whether a substring is present in another string, ignoring casing differences.

These methods can be categorised into *string validation* and *string processing* routines. String validation routines take one or more string inputs and return true or false, where string processing routines take one or more string inputs and return a string output. Methods *containsChar*, *containsIgnoreCase*, *isGroup*, *isHostNameLabel*, *isSimpleAddress*, *isValidUsername*, *lesseqString*, *needsUrlEncoding*, and *validateEmail* are considered as string validation routines, and methods *capitalise*, *composeName*, *encodeClassName*, *getClassName*, *protectQuotations*, *stripName*, *toCamel* and *translate* are classified as string processing routines.

5.4.2 Generating String Test Inputs

The first phase of the experiment involved generating string test inputs for each method using a conventional search-based approach and the language model informed approach. The (1+1) Evolutionary Algorithm [119, 118] was used to attempt full branch coverage within the maximum allowance of $100,000$ fitness evaluations for each branch. The search mechanism was repeated 30 times using an identical set of random seeds for each approach in order to enlarge the test data's sample size, and therefore avoid the potential source of bias.

The (1+1) EA is the simplest variation of a Evolutionary Algorithm which operates based on a continuous process of recombination, mutation, and selection to produce individuals that are progressively evolved. The (1+1) EA, similar to Hill Climbing uses only one current point in the search space, but instead of selecting the next best point in its neighbourhood, it modifies through random changes called mutations. If the modified value has an improved fitness than the original value, it replaces the original. The full description of (1+1) EA was presented in Section 2.4.4.

The conventional approach may terminate before the specified limit of $100,000$ fitness evaluations if the test data covering the branch is located. The language model approach, however, continues optimising the branch-covering string input to achieve the best language model score.

The IGUANA toolset [80] was used as a search-based framework for generating test inputs. The string representation was specified as an array

of characters, s, of length 30, followed by an additional integer l, which was used to control the string's length. For instance, if $l = 5$, the first five characters of s were used to form a string of length 5. Each character of s was in the ASCII printable range of 32–126. Mutations were made at a probability of $\frac{1}{l+1}$ using the uniform mutation. Test inputs generated using both approaches covered exactly the same branches, since the language model approach is identical to the conventional approach up until a branch is covered – at which point it begins to improve the readability of the string input. During test data generation, 100% branch coverage was achieved for all methods, except for one infeasible branch in *validateEmail*.

5.4.3 Test Input Selection

Test data generated using each approach for the 17 consisted of large number test cases due to the 30 runs. This amount varied for each method due to their various number of of branches. To create a uniform experimental setup, a fixed number of (30) test inputs were selected for each program from each approach. This selection mechanism was implemented using the randomisation scheme employed in previous experiment. The scheme was set to select a total of 30 diverse test inputs generated using each approach for each method . This allocated each method with a total of 60 test inputs, to be evaluated by human subjects at the end of the evaluation task.

5.4.4 Human Study Protocol

This phase of the empirical study was concerned with assessing the time human subjects required to manually evaluate string inputs generated by each approach. This process was automated as a web application similar to the TCEvaluator. The application initially presented a brief description about the study, and requested the participants to specify their level of education and their field of expertise. It then displayed a paragraph of text describing the operation of one of the 17 methods selected at random. In addition, for the *lesseqString* method, supplementary information was supplied regarding the ASCII codes.

The application then displayed 8 questions to the participants to answer.

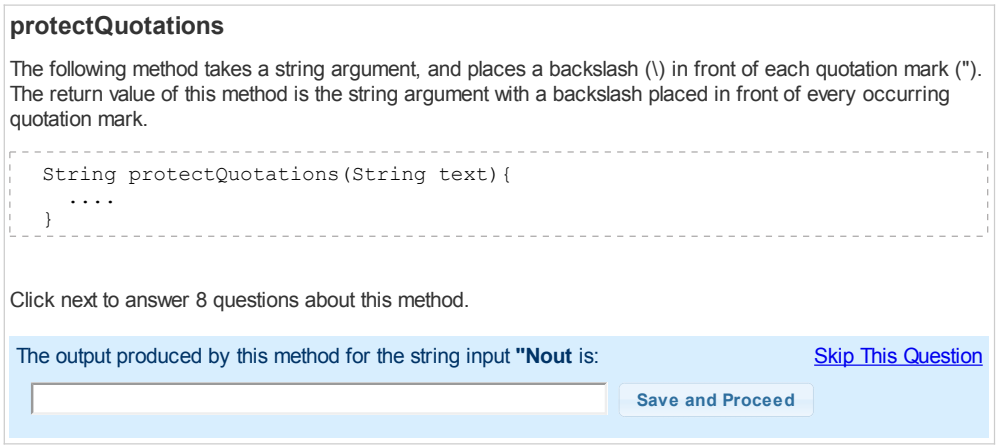


Figure 5.3: Example of a question in the TCEvaluator

In order to familiarise the participant with the case study, the first two questions were assigned as practice questions. For each question, a test input was selected randomly from a pool of 60 inputs, requesting the participant to simply provide the expected output for the given method. This would be a boolean value (i.e. the user would be expected to enter "true" or "false") for a method in the *string validation* category. For a *string conversion* method, the participant would be expected to enter a string return value.

The time taken from the presentation of each question to the participant clicking "next" and having entered their response was recorded, with the response logged internally as "correct" if it matched the actual return value of the method. Once the participant had completed their their evaluation task for a case study method, they were not allowed to go back and change any answers or re-take it with a different set of questions. Figure 5.3 shows a screenshot of the application with an example question.

5.4.5 Participant Selection

Due to the limited number of volunteer students, the participants for this study were only included crowd-workers recruited via CrowdFlower. These participants were required to have some level of self-reported experience in computer programming. As previously described in the last two chapters, one of the major risks in crowd-sourcing studies was the quality of data

due to the diversity of online workers, and their unknown level of expertise, which would allow people to provide arbitrary answers just to gain money. To avoid this, participants who evaluated less than half of the 8 questions correctly were considered as ineligible and their responses were discarded from the analysis of the data presented in Section 5.5.

5.4.6 Usable Judgements

To reduce the potential "learning effects", the answers of the first two practice questions were discarded from the data analysis. This resulted in 250 responses involving inputs generated using each approach for each method (500 for each method in total) for analysis of results presented in Section 5.5.

5.4.7 Research Questions

The research questions to be answered by the empirical study are as follows:

RQ 1. Test Data Language Model Score. This research question investigates whether incorporating the search-based test data generation process with a language model can significantly improve the language model scores of generated strings, and if so, by how much.

RQ 2. Test Data Accuracy Score. This research question checks whether the use of a language model in the search-based test data generation process can significantly increase the accuracy score of generated test inputs. Accuracy score in this context refers to the percentage of participants who accurately evaluated test inputs generated using each approach. To answer this question, the accuracy score for test inputs generated using the language model will be computed and compared to those generated without the use of the language model for each case study.

RQ 3. Test Data Cognition Time. This research question inspects whether incorporation of a language model into the search-based test data generation process can significantly reduce the evaluation time of the resulting test inputs. To answer this research question, the time required for

participants to enter the correct outputs for the inputs generated using each approach will be compared. This will be performed on two selections of data: all the collated responses, and only the responses in which participant provided the correct expected output (i.e correct judgements).

RQ 4. Test Data Mutation Score. This research question investigates whether the use of a language model in search-based test data generation will have significant effects on the fault-finding capability of generated test inputs.

5.5 Experimental Results

This section discusses the results obtained from the human empirical study, inspecting each research question.

RQ1 - Test Data Language Model Score

The average language model scores were computed for all non-empty strings generated using both the conventional and the language model approach. The results, demonstrated as a bar chart in Figure 5.4, revealed that the probability scores of strings generated using the conventional approach were significantly lower than those generated using the language model approach. This improvement was at the very least doubled (i.e *ContainsIgnoreCase*), and in the best cases varied up to several orders of magnitude. Highest language model scores corresponded to case studies such as *containsChar* and *lesseqString*. These methods implicate few constraints with regards to the presence of certain characters that generally do not conform to elements of natural words – for instance the '`@`' symbol in email addresses.

In answer to this research question, the evidence suggests that the language model can successfully be incorporated into an evolutionary algorithm to generate strings with higher language model scores.

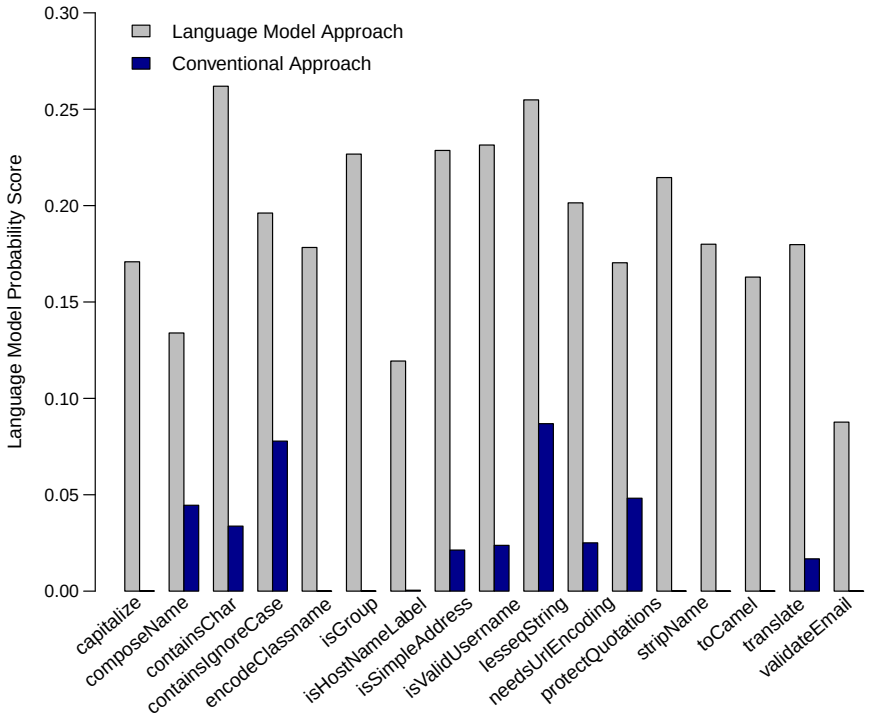


Figure 5.4: Average language model scores for string test inputs generated using the conventional search-based approach in comparison to those generated using the language model approach

Table 5.2: The results of the Fisher's exact test on the percentage of correct judgements obtained (accuracy score) for test inouts of each approach. A p-value in bold face indicates the cases where the use of language model approach had possitive effects on accuracy score.

Case Study	**Lang.** (%)	**Conv.** (%)	***p*-value**
capitalize	74.4	79.6	0.635
composeName	80.4	82.0	0.894
containsChar	94.0	90.0	0.747
containsIgnoreCase	85.6	84.8	0.948
encodeClassname	**97.2**	**74.8**	**0.048**
getClassName	83.6	80.4	0.790
isGroup	95.6	96.8	0.949
isHostNameLabel	87.6	86.4	0.948
isSimpleAddress	94.4	90.4	0.747
isValidUsername	87.2	94.0	0.604
lesseqString	78.4	78.0	1.000
needsUrlEncoding	95.6	96.8	0.949
protectQuotations	88.0	84.8	0.793
stripName	**90.0**	**59.2**	**0.003**
toCamel	**90.4**	**59.2**	**0.003**
translate	88.4	84.8	0.793
validateEmail	72.0	89.6	0.108

RQ2 - Test Data Accuracy Score

To obtain the accuracy score for each type of test data, the percentage of test inputs generated by each approach, for which the participant correctly entered the output were computed for each case study. A Fisher's exact test was next performed on the numbers of questions that were correctly answered at a confidence level of 95%. This test was performed to compare the accuracy rate of test inputs generated using each approach. The results of this test (presented in Table 5.2) showed no significant differences between the language model approach and conventional approach for the majority of case studies. However, three of the string processing methods such as *encodeClassName*, *stripName* and *toCamelCase* did reveal a significant improvement when using the language model. In addition, there was no evidence to indicte that inputs generated using the language model approach reduced accuracy score significantly in any of the methods under consideration.

In answer to this research question, therefore, the analysis suggests that the language model can indeed improve the accuracy of test input evaluation for certain classes of programs.

RQ3 - Test Data Cognition Time

To answer this research question, the time participants required to answer each question was recorded. The average time required for all judgements and the correct judgements were computed saparately. A correct judgment was defined as the one in which the participant entered the correct output for the given input. Average in this context was defined as the mean of timing dataset. The results are represented in Table 5.3.

To investigate statistical significance on mean times required to evaluate test inputs generated using each approach, the Wilcoxon rank-sum test was performed at a confidence level of 95%. This test was performed on both selections of the data: mean times for all judgments and mean times only for correct judgments. For both of these selections, the average cognition time for inputs generated using the language model approach revealed to be significantly lower than those generated using the conventional approach for

Table 5.3: Cognition time for seeded and unseeded test data based to (a) all judgements, and (b) correct judgements only

(a) All judgements

Case Study	**Lang.** (s)	**Conv.** (s)	***p*-value**	$\hat{A}_{12}$
capitalize	14.1	16.6	0.820	0.506
composeName	20.6	21.7	0.596	0.514
containsChar	**9.1**	**14.3**	**< 0.001**	*** **0.279**
containsIgnoreCase	10.4	10.2	0.877	0.496
encodeClassname	**15.1**	**40.3**	**< 0.001**	*** **0.100**
getClassName	**10.5**	**19.5**	**< 0.001**	** **0.328**
isGroup	**6.1**	**6.8**	**0.006**	* **0.429**
isHostNameLabel	**6.9**	**8.9**	**< 0.001**	* **0.414**
isSimpleAddress	**7.9**	**13.9**	**< 0.001**	** **0.350**
isValidUsername	6.2	5.5	0.946	0.502
lesseqString	**14.0**	**24.0**	**< 0.001**	* **0.407**
needsUrlEncoding	8.6	8.8	0.114	0.541
protectQuotations	**13.1**	**15.8**	**< 0.001**	* **0.380**
stripName	**20.4**	**42.7**	**< 0.001**	*** **0.194**
toCamel	**16.4**	**33.6**	**< 0.001**	*** **0.193**
translate	15.5	25.6	0.089	0.456
validateEmail	9.8	7.5	0.136	0.539

(b) Correct judgements only

Case Study	**Lang.** (s)	**Conv.** (s)	***p*-value**	$\hat{A}_{12}$
capitalize	11.6	12.1	0.333	0.529
composeName	19.2	19.9	0.579	0.516
containsChar	**9.2**	**14.6**	**< 0.001**	*** **0.279**
containsIgnoreCase	10.3	9.4	0.707	0.511
encodeClassname	**15.0**	**35.8**	**< 0.001**	*** **0.089**
getClassName	**10.8**	**16.8**	**< 0.001**	** **0.329**
isGroup	**6.1**	**6.6**	**0.014**	* **0.435**
isHostNameLabel	**7.0**	**9.0**	**0.005**	* **0.422**
isSimpleAddress	**7.4**	**13.6**	**< 0.001**	** **0.335**
isValidUsername	6.2	5.3	0.756	0.508
lesseqString	**13.7**	**26.8**	**< 0.001**	* **0.367**
needsUrlEncoding	8.8	8.9	0.124	0.541
protectQuotations	**13.0**	**15.7**	**< 0.001**	* **0.371**
stripName	**20.3**	**45.0**	**< 0.001**	*** **0.134**
toCamel	**16.4**	**32.9**	**< 0.001**	*** **0.178**
translate	15.8	25.3	0.552	0.483
validateEmail	8.4	7.5	0.284	0.531

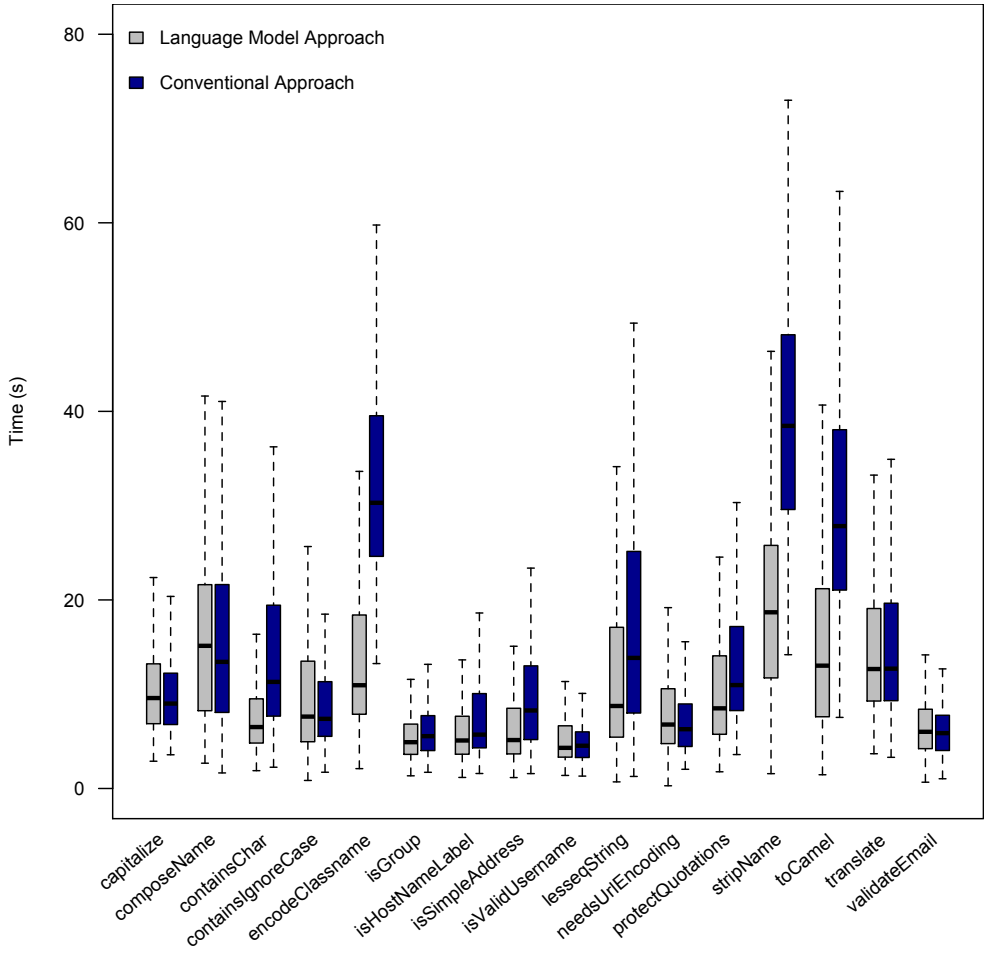


Figure 5.5: Box and whisker plots displaying the timing distribution for performing "correct" evaluations on test inputs of each approach on each method. The centre line represents the median and the box the distribution of the data between the upper and lower quartiles. The upper and lower whiskers represent the minimum and maximum value (excluding outliers).

10 of the 17 case studies.

Figure 5.5 shows box and whisker plots of the times recorded for correct judgements. The plots show particular superiority of the language model's ability to produce shorter evaluation times for certain case studies, including *containsChar*, *encodeClassName*, *stripName* and *toCamel*. This was confirmed by the computation of effect sizes, using Vargha and Delaney's $\hat{A}_{12}$ statistic [117], and recorded in Table 5.3. According to the information provided in Vargha and Delany's paper an effect size is categorised as *large*, *medium* or *small* when values are less than 0.29, 0.36 and 0.44 respectively. Based on this classification, cognition times for test inputs generated using the language model approach had large effect sizes in 4 case studies: *containsChar*, *encodeClassName*, *stripName* and *toCamelCase*. The latter three had revealed significant improvements in accuracy as response to the previous research question. The results obtained for *isSimpleAddress* and *getClassName* had medium effect size, while *isGroup*, *isHostNameLabel*, *lesseqString* and *protectQuotations* experienced medium effect size.

There was no significant difference on the remaining 7 case studies. Methods *capitalize*, *composeName* and *translate* were classified as string processing methods that did not require their string inputs to be fully comprehended by testers to provide outputs. *capitalize* merely required the tester to check the first character of the string. *composeName* purely outputs one of its two arguments or the concatenation of both, without requiring the two string arguments to actually be read. Method *translate*, in a similar style to *capitalize*, only required examination of the first character of the string. This can explain why no significant difference was found in terms of evaluation times for the two approaches in these string processing methods.

containsIgnoreCase was classified as a string processing method, which did require its string inputs to be fully comprehended by testers. There was however no significant differences between the two approaches in terms of evaluation time of test inputs generated for this method. The majority of the test cases generated for this method using both approaches included an empty string for the second input variable. Since this method checks the presence of the second string argument in the first, the empty strings generated for the second argument would result in trivially returning false

in most cases. This accounts for the lack of difference in times recorded for both language model and conventional approaches.

Finally, the test data generated for 3 of the string validation routine: *isValidUsername*, *needsUrlEncoding* and *validateEmail* revealed to present more straightforward tasks for the human participants, and thus the usage of a language model failed to produce any significant differences in terms of relevant evaluation time.

RQ4 - Test Data Fault Finding Capability

To answer this research question, the mutation system for Java programs, MuClipse [7] was used to assess the fault-finding capability of test inputs generated using each approach. The Fisher exact test was performed to determine the statistical significance with confidence level set to 95%. Table 5.4 presents the mutation scores for test cases generated using each approach, with p-values displayed in the last column. Each test set is compromised of all diverse test cases generated using each approach for each method. Table 5.5 shows the number of generated mutants, and the number of diverse test cases of each approach for each program.

As evident from Table 5.4, test inputs generated using the language model approach have slightly lower mutation scores for programs *isSimpleAddress*, *needsUrlEncoding*, *stripName*, and *translate*. This difference however is not statistically significant as none of the corresponding p-values are within the significance threshold (< 0.05). Mutation score for *composeName* and *encodeClassname* is undefined as MuClipse failed to generate any mutants for these methods. This is merely because the traditional operators only modify syntactic features such as arithmetic operators, unary logic, etc. These methods do not contain any of such features.

In response to this research question, the analysis of results indicate that, the mutation score remains unchanged for test inputs generated using each approach, and thus the use of the language model has no significant effects on fault-finding capabilities of the test suites.

Table 5.4: Mutation sccore of test cases generated using language model approach and the conventional approach. Mutation score in this context refers to the percentage of mutants that each test set can detect.

Project	**Method**	**Lang.** (%)	**Conv.** (%)	***p*-value**
Bots'n'Scouts	lesseqString	83	83	1.0
CodeHaggis	toCamel	19	19	1.0
Daikon	getClassName	100	100	1.0
	protectQuotations	100	100	1.0
Germoglio	translate	83	91	0.6
Jake	isValidUsername	100	100	1.0
JavaMail	isSimpleAddress	79	81	0.9
	isGroup	73	73	1.0
JOX	stripName	91	93	1.0
Muffin	containsChar	94	94	1.0
OpenJDK	isHostNameLabel	100	100	1.0
PuzzleBazar	composeName	0	0	1.0
Rife	validateEmail	100	100	1.0
	capitalize	50	50	1.0
	encodeClassname	0	0	1.0
	needsUrlEncoding	82	86	0.8
Subsonic	containsIgnoreCase	100	100	1.0

Table 5.5: Shows the number of diverse test cases generated using each approach, the total number of generated mutants, and the number of mutants that were detected (killed) by these test cases.

Project	**Method**	**Diverse**		**Mutants**	**Killed**	
		Lang.	**Conv.**		**Lang.**	**Conv.**
Bots'n'Scouts	lesseqString	60	60	6	5	5
CodeHaggis	toCamel	66	57	139	27	27
Daikon	getClassName	56	59	15	15	15
	protectQuotations	60	52	36	36	36
Germoglio	translate	58	60	137	114	126
Jake	isValidUsername	105	92	10	10	10
JavaMail	isSimpleAddress	50	55	96	76	78
	isGroup	89	90	26	19	19
JOX	stripName	59	40	73	67	68
Muffin	containsChar	111	63	19	18	18
OpenJDK	isHostNameLabel	81	69	110	110	110
	composeName	91	90	0	0	0
PuzzleBazar	validateEmail	352	687	466	466	466
Rife	capitalize	70	60	20	10	10
	encodeClassname	30	30	0	0	0
	needsUrlEncoding	118	72	153	126	133
Subsonic	containsIgnoreCase	93	90	2	2	2

5.6 Threats to Validity

This section discusses the threats to validity associated with the human empirical study.

As described in Section 3.5 and Section 4.4 in previous chapters, the main threat to internal validity in empirical studies regarding search-based test data generation is the stochastic nature of the meta-heuristic search algorithms. The chosen scheme for mitigating this threat was to perform the test data generation process using a sufficiently large data sample. The employed search method (i.e. (1+1) EA) in this study was thus executed 30 times using each approach on each program. This provided a large pool of test data from which to draw observations, ensuring sample means were normally distributed. This was extensively explained in Section 5.4.2.

One of the major threats to the external validity is the selection of case studies, and the possibility of obtaining exclusive results that may not hold in practice (for other programs). Due to the diversity of real world programs it is impractical to sample a large set that captures all the possible characteristic of all programs. In order to mitigate these risks, the case studies were selected from various real world open source projects. In order to facilitate capturing various aspects of functionality regarding string inputs, only methods with string arguments were used. This drew a total of 17 methods which were selected from 12 open source project. These issues were fully discussed in Section 5.4.

Another source of potential bias is concerned with the use of the crowd-sourcing website, and the selection of participants. As reviewed in Section 4.2.3, crowd-sourcing platforms such as CrowdFlower are commonly open to users mis-reporting experience levels or performing tasks randomly in order to earn money. To mitigate this, participant selection was performed based on an independent metric, such as "all participants with $\geq 50\%$ accuracy". Based on this metric, only a subset of participants who correctly evaluated 50% or more of the test inputs were considered in the analysis of the results. This was broadly explained in Section 5.4.5.

As previously discussed, a major source of bias in human empirical studies is the "learning" or "training" effects. To alleviate these effects in this

experiment the first two responses of each participant were removed from consideration in the analysis of the results. Steps taken to avoid fatigue effects was to keep the form of the questions as simple and limit the number of questions to 8. The analysis of the results revealed that each participant spent approximately 3 minutes on a questionnaire on average. This indicates that the study was relatively trivial and thus unlikely to be subject to fatigue effects. However, the main step to discard any bias from learning and fatigue effects was to randomise the questions and the order in which they appeared to the participants. The test inputs that formed the basis of each question were selected at random from a large pool of inputs, and these were selected at random from the overall set of inputs generated using each stochastic approach. These issues were fully discussed in Sections 4.2.1 and 5.4.4.

Similar to previous experiments, there are potential threats to construct validity since the performance of each approach was mainly assessed based on evaluation time of test data generated using each approach. This essentially originates from the nature of the human empirical study, in which, the experience and skills of the participants and their familiarity with the case studies are unknown. These were handled by randomising the order and the type of the questions for each participant.

Finally, the Fisher's exact test and the Wilcoxon rank-sum test were used to test the statistical significance. Vargha and Delaney's $\hat{A}_{12}$ statistic was used to test the effect size.

5.7 Conclusions

This chapter introduced a new approach to automatic test data generation for string inputs. In this approach, a statistical language model was incorporated into the search-based test data generation process to optimise the readability of string test inputs. The empirical study involving human participants revealed that the string inputs generated using this approach were significantly quicker to evaluate in several case studies. The accuracy of test input evaluation were also shown to be improved in certain cases. Mutation analysis was performed to assess the effects of the language model

approach on mutation score. The outcome revealed that this approach had no significant changes on the fault-finding capabilities of the resultant test suites.

Chapter 6

Conclusions and Future Work

6.1 Summary of Achievements

This thesis explored various aspects of human oracle costs, investigating and establishing methods that could effectively reduce these costs. The thesis focused on minimising qualitative oracle expenses by increasing the readability of automatically generated test data while maintaining the fault-finding effectiveness constant. The main hypotheses of this thesis were outlined in Chapter 1, while each hypothesis was then inspected individually in subsequent chapters.

This chapter summarises the discussion of each hypothesis, outlining the empirical studies performed to determine the validity of each hypothesis. It then presents the main contributions of the thesis. This is followed by a discussion on limitations of the research, and avenues for future investigation.

6.1.1 Hypotheses

Hypothesis 1 *Seeding the search-based test data generation process with human-supplied test inputs can produce test data with higher branch coverage, and without any detrimental effects on fault-finding effectiveness.*

The empirical work in Chapter 3 addressed this hypothesis by performing

an empirical study in which samples of test cases were collated from human subjects for the Java method listed in Table 3.1. The human-supplied test cases were then used as seeds to commence the search-based test input generation process. The results of this empirical study (presented in Section 3.4) revealed that the application of the seeded search-based approach resulted in higher branch coverage for a majority of methods with string inputs. There was no evidence to indicate that the seeded approach had detrimental effects on fault-finding capabilities in any of the methods under consideration.

Hypothesis 2 *Seeding the search-based test data generation process with human-supplied test inputs can produce readable test data that are less time-consuming and error-prone for manual evalution.*

The empirical work presented in Chapter 4 addressed this hypothesis by performing an empirical study in which human subjects were recruited to manually evaluate test cases generated using both the seeded and unseeded search-based approaches by hand, while being timed. Human subjects were expected to provide the correct outputs of a Java method for a set of test cases. The main purpose of this empirical study was to assess the time human subjects required to manually evaluate test cases of each approach, and to investigate whether test data generated using the seeded approach were less time consuming and less error-prone to evaluate. The results of this study (presented in Section 4.3) indicated that the use of the seeded search-based approach can have both positive and negative effects on test data evaluation costs (i.e time and accuracy) depending on the classes of programs.

Hypothesis 3 *Incorporating the search-based test data generation process with a statistical language model can produce more readable test data for string inputs, which are less time-consuming and error-prone for manual evalution.*

The empirical work presented in Chapter 5 addressed this hypothesis by introducing a new approach in which the search-based test data was

incorporated with a language model to encourage generation of readable values for string inputs. This chapter presented an empirical assessment of the technique, in which human subjects were recruited and requested to manually evaluate test data generated using both the language model approach and the conventional search-based approach while being timed. The time and accuracy of human subjects in evaluating test cases of each approach were assessed and compared. The effects of this approach on test data fault-finding effectiveness was also inspected. The results revealed that test data generated using the language model approach were less time consuming to evaluate in several cases, and in certain cases, the accuracy of test input evaluation with respect to outputs was also improved, with no detrimental effects on fault-finding capabilities.

Due to frequent unavailability of automated oracles in software engineering practice, this work therefore revealed an important bearing on lowering the costs of human involvement in the testing process as an oracle.

6.1.2 Contributions of this Thesis

1. The results of the empirical study in which a seeded search-based approach was implemented and compared against a conventional unseeded approach for generating branch-covering and fault-detecting test inputs, revealing cases where the seeded approach outperformed the unseeded approach in terms of branch coverage, efficiency, and fault- finding effectiveness.

2. The results of a human study in which test data generated using both the seeded and unseeded search-based approaches were evaluated by human subjects, revealing cases where seeded test data was both less time consuming and less error-prone to manually evaluate by human subjects.

3. Introduction of a technique for incorporating the automatic test input generation process with a statistical language model to generate readable branch-covering string inputs.

4. The results of a human study in which the language model technique

was compared with a conventional, non-informed approach, revealing cases where test inputs generated using the language model approach were both less time consuming and error-prone to manually evaluate by human subjects.

6.2 Summary of Future Work

The use of the language model approach demonstrated a major difference in readability of the resultant test data and subsequently a significance reduction in test data cognition time. This opens a venue for further research and investigations that can widen the applicability and scalability of the approach at minimal costs. This section outlines restrictions and limitations of the language model approach, and describes how some of these can be resolved.

6.2.1 Investigating Various Seeding Schemes

The first experiment of this thesis involved investigating a seeding technique in which seeds were collated directly from humans via a crowd-sourcing platform. As discussed in Chapter 3, only the correct values obtained from the crowd were selected and used as seeds. The term "correct values" were assigned to the pairs of inputs and outputs that correlated correctly against the methods description. As a results, the pairs of values that did not match the methods description were eliminated from the study. As explained in Section 3.5, this step was an essential procedure to prevent unauthorised users from gaming the system for money.

In practise, the data used for seeding may not be pre-filtered, and thus, the choice of filtering can add a potential threat to validity. Future work is required to investigate this, and to explore how different sources of seeds and the quality of the seeded data can affect the approach.

6.2.2 Managing Fault-Finding Capability

As discussed previously, production of readable test data using the language model approach had no significant impact on the relevant fault-finding fac-

tor. However, in large scale software systems, locating critical defects may essentially depend on the presence of unnatural and arbitrary looking test inputs. It was discussed by McMinn et al [82] that less readable test cases are in fact more likely to reveal software faults. This implies that more diverse or randomly generated test cases are not without a purpose, and that these are required for revealing software failures.

As a direction for future work, large scale empirical studies involving Mutation Analysis should be performed to identify the relationship between readability and fault-finding effectiveness. In any case, the multi-objective search-based procedures can be applied to determine the optimal trade-off between reduced human oracle costs and increased fault-finding capabilities. Multi-objective search has previously been employed in search-based test data generation [52]. Given a certain time budget in which to perform testing, readable test cases may be prioritised in order to reduce oracle checking time and thus increase the number of test cases that may be considered. Given unlimited time, more faults may be detected with test cases produced using the conventional techniques.

6.2.3 Improving Readability

While it is important to test programs with unrealistic inputs, it is also important to test them with natural instantly-readable values. This is mainly due to the following reasons:

1. Readable inputs are easier to comprehend by human testers and can therefore reduce oracle checking time.

2. Faults found by readable test cases are more likely to be prioritised for fixing. According to Bozkurt et al [24], a fault-revealing test case that rarely or never occurs in practice is unlikely to attract the attention of a tester who is preoccupied with various tasks such as bug reports.

3. Readable test inputs can represent important corner cases [24] which are rarely generated using the conventional search-based techniques.

Although the language model approach can generate more readable strings (such as "*s@prerereandes.Nouthin*"for an email address) - it is however in-

Table 6.1: Samples of strings generated using the language model approach, and a set of realistic examples displayed in the last column.

Method	**LangModel**	**Realistic**
toCamel	inererof_yo	persian_cat
protectQuotations	“Nout	He said “Hi”
isGroup	in:theandes;	recipient-list: tom@ymai.com;
validateEmail	heres@pe.HALOL	James@gmail.com
capitalize	hinthere	oxford

capable of producing realistic strings that represent real-world entities or natural words that exist in the dictionary. Table 6.1 lists samples of strings generated using the approach and a realistic example for each case.

Further work is required to investigate and improve the readability of test data generated using the language model approach. This involves exploring various types of texts employed to train the language models, and to investigate how these can increase the readability of strings produced. The text used to train the current language model was an electronic version of the classic novel Moby Dick [85], consisting of 215,133 words and 1,235,150 characters. Larger corpora containing certain types of text may be more suitable for certain domains of programs – For instance books regarding different programming languages that contain samples of source codes or even an electronic version of a dictionary that contains all the existing words in the language.

6.2.4 Test Input Generation for Various Data Types

The language model is specifically designed for test data generation for string data types. It is however not applicable on string variables that represent diverse and complex types of real-world data such as serial numbers, registry codes, uniform resource locators and international banking digits.

To circumvent this issue, the language model technique can be combined with the web-query approach [84, 105] (proposed by McMinn et al) to form an integrated application that can generate readable test data for all data

types. As described in the literature review, Section 2.4.6, the use of the web-query approach requires the program to have useful identifiers that can be reformulated into web search queries. Combination of this approach with the language model technique promises to provide the means to automatically generated readable inputs for different programs.

The key purpose of future work is further reduction of human oracle costs. These venues should therefore be explored to efficiently improve the applicability and scalability of the language model approach, and to assess its consequential impact on manual evaluation costs.

References

[1] Amazon Mechanical Turk website. https://www.mturk.com/mturk/welcome.

[2] Android code style guidelines. http://source.android.com/source/code-style.html#write-short-methods.

[3] Crowd Guru website. http://www.crowdguru.de/.

[4] CrowdFlower website. http://crowdflower.com/.

[5] The Eclipse foundation open source community website. http://www.eclipse.org/.

[6] Microsoft research. http://research.microsoft.com/.

[7] Muclipse. http://muclipse.sourceforge.net/.

[8] Software observatory homepage. http://observatory.group.shef.ac.uk/.

[9] Object-z: A specification language advocated for the description of standards. *Computer Standards Interfaces*, 17:511 – 533, 1995.

[10] *Empirical Methods and Studies in Software Engineering, Experiences from ESERNET*, volume 2765 of *Lecture Notes in Computer Science*. Springer, 2003.

[11] J. Abrial, M. Lee, D. Neilson, P. Scharbach, and I. Srensen. The b-method. In *VDM '91 Formal Software Development Methods*, volume 552, pages 398–405.

[12] David H. Ackley. *A connectionist machine for genetic hillclimbing.* Kluwer Academic publishers, Norwell, MA, USA, 1987.

[13] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 348–357, Washington, DC, USA, 1993. IEEE Computer Society.

[14] S.G. Alawneh and D.K. Peters. Specification-based test oracles with junit. In *Electrical and Computer Engineering (CCECE), 2010 23rd Canadian Conference on*, may 2010.

[15] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *International Conference on Automated Software Engineering (ASE)*, pages 3 –12, November 2011.

[16] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators: Research articles. *Software Testing, Verification and Reliability*, 16(3):175–203, September 2006.

[17] James H. Andrews and Yingjun Zhang. General test result checking with log file analysis. *IEEE Transactions on Software Engineering*, 29:634–648, July 2003.

[18] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608 –624, aug. 2006.

[19] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 298–308, Washington, DC, USA, 2009. IEEE Computer Society.

[20] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. In *Proceedings of the 2010 Third*

International Conference on Software Testing, Verification and Validation, ICST '10, pages 205–214, Washington, DC, USA, 2010. IEEE Computer Society.

[21] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 14–21, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[22] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. *SIGSOFT Software Engineering Notes*, 27(4), July 2002.

[23] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT - A formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–244. ACM Press, 1975.

[24] Mustafa Bozkurt and Mark Harman. Automatically generating realistic test input from web services. In *SOSE'11: Proceedings of the 6th IEEE Symposium on Service-Oriented System Engineering*, pages 13–24, December 2011.

[25] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Comparative assessment of testing and model checking using program mutation. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society.

[26] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 86–95, Washington, DC, USA, 2004. IEEE Computer Society.

[27] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: a system for selective regression testing. In *Proceedings of the 16th In-*

ternational Conference on Software Engineering, ICSE '94, pages 211–220, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[28] J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.

[29] John A. Clark, Haitao Dan, and Robert M. Hierons. Semantic mutation testing. *IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 100–109, 2010.

[30] Haitao Dan and Robert M. Hierons. Semantic mutation analysis of floating-point comparison. *International Conference on Software Testing, Verification, and Validation*, 0:290–299, 2012.

[31] Haitao Dan and Robert M. Hierons. Smt-c: A semantic mutation testing tools for C. *International Conference on Software Testing, Verification, and Validation*, 0:654–663, 2012.

[32] Martin D. Davis and Elaine J. Weyuker. Pseudo-oracles for non-testable programs. In *ACM '81: Proceedings of the ACM '81 conference*, pages 254–257, New York, NY, USA, 1981. ACM.

[33] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[34] Julie S. Downs, Mandy B. Holbrook, Steve Sheng, and Lorrie Faith Cranor. Are your participants gaming the system?: screening mechanical turk workers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2399–2402, New York, NY, USA, 2010. ACM.

[35] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+ 1) evolutionary algorithm. 276(1-2), April 2002.

[36] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27:213–224, 2001.

[37] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Deember 2007.

[38] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[39] J. Ferrante, K. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[40] John S. Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef. *Vienna Development Method*. John Wiley and Sons, Inc., 2007.

[41] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

[42] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 121 –130, April 2012.

[43] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 177–187, New York, NY, USA, 2012. ACM.

[44] Zachary P. Fry and Westley Weimer. A human study of fault localization accuracy. In *Proceedings of the 2010 IEEE International Con-*

ference on Software Maintenance, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[45] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. *ACM SIGPLAN Notices*, 40(6):213–223, June 2005.

[46] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. 40(6):213–223, June 2005.

[47] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[48] M. Harman. Open problems in testability transformation. In *IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW)*, pages 196–209, 2008.

[49] M. Harman, A. Baresel, D. Binkley, R. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper. Testability transformation - program transformation to improve testability. In *Formal Methods and Testing, Lecture Notes in Computer Science*, volume 4949, pages 320–344. Springer-Verlag, 2008.

[50] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener. The impact of input domain reduction on search-based test data generation. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2007)*, pages 155–164, Cavat, near Dubrovnik, Croatia, 2007. ACM Press.

[51] M. Harman, Sung Gon Kim, K. Lakhotia, P. McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 182 –191, April 2010.

[52] M. Harman, K. Lakhotia, and P. McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the Genetic and*

Evolutionary Computation Conference (GECCO 2007), pages 1098–1105, London, UK, 2007. ACM Press.

[53] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 73–83, London, UK, 2007. ACM Press.

[54] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*, 36:226–247, 2010.

[55] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[56] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 837–847, 2012.

[57] Douglas Hoffman. Heuristic test oracles. *Quality Engineering Magazine*, 1999.

[58] Mike Holcombe. *Starting an XP Project*, pages 73–117. John Wiley and Sons, Inc., 2008.

[59] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[60] W.E. Howden. *Introduction to the Theory of Testing.* IEEE Computer Society Press, 1978.

[61] X.B. Hu and E. Di Paolo. An efficient genetic algorithm with uniform crossover for the multi-objective airport gate assignment problem. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 55 –62, September 2007.

[62] Kenneth A. De Jong and William M. Spears. A formal analysis of the role of multi-point crossover in genetic algorithms. *Annals of Mathematics and Artificial Intelligence*, 1992.

[63] D. Jurafsky and J. Martin. *Speech and Language Processing*. Pearson, second edition, 2009.

[64] J. S. Karn, S. Syed-Abdullah, A. J. Cowling, and M. Holcombe. A study into the effects of personality type and methodology on cohesion in software engineering teams. *Behav. Inf. Technol.*, pages 99–111, March 2007.

[65] John Karn and Tony Cowling. A follow up study of the effect of personality on the performance of software engineering teams. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical Software Engineering*, pages 232–241, New York, NY, USA, 2006.

[66] J.S. Karn and A.J. Cowling. An initial observational study of the effects of personality type on software engineering teams. *IEE Seminar Digests*, 2004(920):155–164, 2004.

[67] S. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoust, Speech, Signal Process*, 35(5):400–401, 1987.

[68] Sunwoo Kim, John A. Clark, and John A. McDermid. Investigating the effectiveness of object-oriented testing strategies with the mutation method. *Software Testing, Verification and Reliability*, 11:207–225, 2001.

[69] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[70] Aniket Kittur, Ed H. Chi, and Bongwon Suh. Crowdsourcing user studies with mechanical turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 453–456, New York, NY, USA, 2008. ACM.

[71] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[72] Marco Laumanns, Lothar Thiele, Eckart Zitzler, Emo Welzl, and Kalyanmoy Deb. Running time analysis of multi-objective evolutionary algorithms on a simple discrete optimization problem. In *Parallel Problem Solving from Nature PPSN VII*, volume 2439 of *Lecture Notes in Computer Science*, pages 44–53. Springer Berlin / Heidelberg, 2002.

[73] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4), April 2007.

[74] A. Lopez. Statistical machine translation. *ACM Computing Surveys*, 40(3), 2008.

[75] Ilya Loshchilov, Marc Schoenauer, and Michèle Sebag. Not all parents are equal for mo-cma-es. In *Proceedings of the 6th International Conference on Evolutionary Multi-criterion Optimization*, EMO'11, pages 31–45, Berlin, Heidelberg, 2011. Springer-Verlag.

[76] H. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, 1999.

[77] Nashat Mansour and Khalid El-Fakih. Simulated annealing and genetic algorithms for optimal regression testing. *Journal of Software Maintenance*, Jnuary 1999.

[78] G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[79] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[80] P. McMinn. IGUANA: Input generation using automated novel algorithms. A plug and play research tool. Technical Report CS-07-14, Department of Computer Science, University of Sheffield, 2007.

[81] P. McMinn, D. Binkley, and M. Harman. Testability transformation for efficient automated test data search in the presence of nesting. In *Proceedings of the UK Software Testing Workshop (UKTest 2005)*, pages 165–182. University of Sheffield Computer Science Technical Report CS-05-07, 2005.

[82] P. McMinn, M. Stevenson, and M. Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *STOV 2010: Proceedings of the 1st International Workshop on Software Test Output Validation*, 2010.

[83] Phil McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1689–1696, New York, NY, USA, 2009. ACM.

[84] Phil McMinn, Muzammil Shahbaz, and Mark Stevenson. Search-based test input generation for string data types using the results of web queries. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 141–150, Washington, DC, USA, 2012. IEEE Computer Society.

[85] H. Melvile. *Moby Dick*. Harper and Brothers, 1851.

[86] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. What test oracle should I use for effective GUI testing? In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 164–173. IEEE Computer Society, October 2003.

[87] E. Miller and W.E. Howden. *Tutorial, software testing & validation techniques*. IEEE Computer Society Press, 1981.

[88] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[89] M. Mitchell, J. Holland, and S. Forrest. When will a genetic algo-

rithm outperform hill climbing? In *Advances in Neural Information Processing Systems 6*, pages 51–58. Morgan Kaufmann, 1993.

[90] Christian Murphy. Using runtime testing to detect defects in applications without test oracles. In *FSEDS '08: Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium*, pages 21–24, New York, NY, USA, 2008. ACM.

[91] Christian Murphy, Kuang Shen, and Gail Kaiser. Automatic system testing of programs without test oracles. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 189–200, New York, NY, USA, 2009. ACM.

[92] Christian Murphy, Kuang Shen, and Gail Kaiser. Using jml runtime assertion checking to automate metamorphic testing in applications without test oracles. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 436–445, Washington, DC, USA, 2009. IEEE Computer Society.

[93] G.J. Myers, C. Sandler, T. Badgett, and T.M. Thomas. *The Art of Software Testing*. Business Data Processing: A Wiley Series. Wiley, 2004.

[94] A. Jefferson Offutt and Ronald H. Untch. Mutation testing for the new century. chapter Mutation 2000: uniting the orthogonal. 2001.

[95] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25:41–61, 1995.

[96] Fabrizio Pastore, Leonardo Mariani, and Gordon Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *ICST*, page to apear. IEEE, 2013.

[97] Yury Pavlov and Gordon Fraser. Semi-automatic search-based test generation. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 777–784, Washington, DC, USA, 2012. IEEE Computer Society.

[98] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, 2000.

[99] H. Pohlheim. Geatbx - genetic and evolutionary algorithm toolbox. `http://www.geatbx.com`.

[100] Christopher Poile, Andrew Begel, and Lucas Layman. Coordination in large-scale software development: Helpful and unhelpful behaviors. 2009.

[101] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, 1992.

[102] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[103] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 218–227, Washington, DC, USA, 2008. IEEE Computer Society.

[104] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE 2005*, pages 263–272. ACM, 2005.

[105] M. Shahbaz, P. McMinn, and M. Stevenson. Automated discovery of valid test strings using dynamic regular expressions collation and tailored web searches. In *Proceedings of the International Conference on Quality Software (QSIC 2012)*, 2012.

[106] Rion Snow, Brendan O'Connor, Daniel Jurafsky, and Andrew Y. Ng. Cheap and fast— but is it good? Evaluating non-expert annotations for natural language tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 254–263, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[107] Fei Song and W. Bruce Croft. A general language model for information retrieval. In *Proceedings of the eighth International Conference on Information and knowledge management*, CIKM '99, pages 316–321, 1999.

[108] Andreas Stolcke. SRILM – an extensible language modeling toolkit. In *Proceedings of ICSLP*, volume 2, pages 901–904, Denver, USA, 2002.

[109] Kathryn T. Stolee and Sebastian Elbaum. Exploring the use of crowdsourcing to support empirical studies in software engineering. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 35:1–35:4, New York, NY, USA, 2010. ACM.

[110] P. Thevenod-Fosse, H. Waeselynck, and Y. Crouzet. An experimental study on software structural testing: deterministic versus random input generation. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 410 –417, June 1991.

[111] David B. Thomas, Wayne Luk, Philip H.W. Leong, and John D. Villasenor. Gaussian random number generators. *ACM Computing Surveys*, 39(4), November 2007.

[112] Chris Thomson and Mike Holcombe. The Sheffield Software Engineering Observatory Archive: Six years of Empirical Data Collected from 73 Complete Projects. Technical report, Department of Computer Science, University of Sheffield, Sheffield, UK, 2009.

[113] Walter F. Tichy. Hints for reviewing empirical work in software engineering. *Empirical Software Engineering Journal*, 5(4):309–312, December 2000.

[114] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Software Engineering Notes, Issue 23, No. 2, Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1998)*, pages 73–81, 1998.

[115] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 285–288, Hawaii, USA, 1998. IEEE Computer Society Press.

[116] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test data generation for exception conditions. *Software - Practice and Experience*, 30(1):61–79, 2000.

[117] Andrs Vargha and Harold D. Delaney. A critique and improvement of the CL common language effect size statistics of mcgraw and wong. *journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[118] Ingo Wegener. Theoretical aspects of evolutionary algorithms. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP 2001), LNCS 2076*, pages 64–78. Springer-Verlag, 2001.

[119] Ingo Wegener. Methods for the analysis of evolutionary algorithms on pseudo-boolean functions. In *Evolutionary Optimization*, volume 48 of *International Series in Operations Research and Management Science*, pages 349–369. Springer US, 2003.

[120] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[121] J. Wegener, H. Pohlheim, and H. Sthamer. Testing the temporal behavior of real-time tasks using extended evolutionary algorithms. In *Proceedings of the 7th European Conference on Software Testing, Analysis and Review (EuroSTAR 1999)*, Barcelona, Spain, 1999.

[122] Elaine J. Weyuker. On testing non-testable programs. *The Computer journal*, 25(4):465–470, 1982.

[123] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[124] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In *5th International Conference on Software Engineering and its Applications*, pages 625–636, Toulouse, France, 1992.

[125] S. Yoo and M. Harman. Test data augmentation: generating new test data from existing test data. Technical Report TR-08-04, King's College London, 2008.

[126] Hao Zhong, Lu Zhang, and Hong Mei. An experimental study of four typical test suite reduction techniques. *Information and Software Technology*, 50(6):534 – 546, 2008.